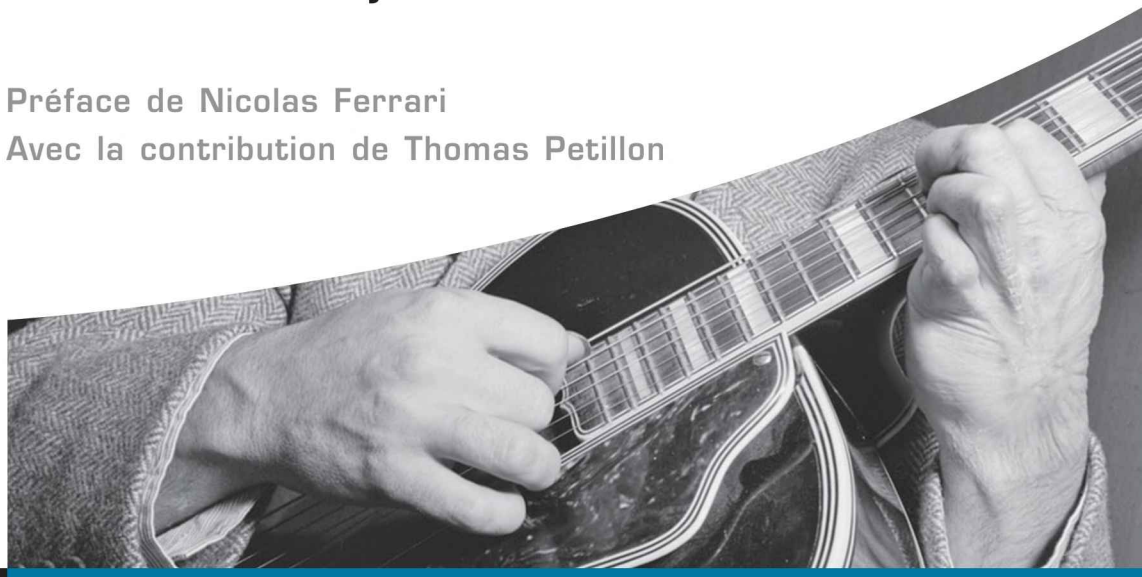
Django avancé

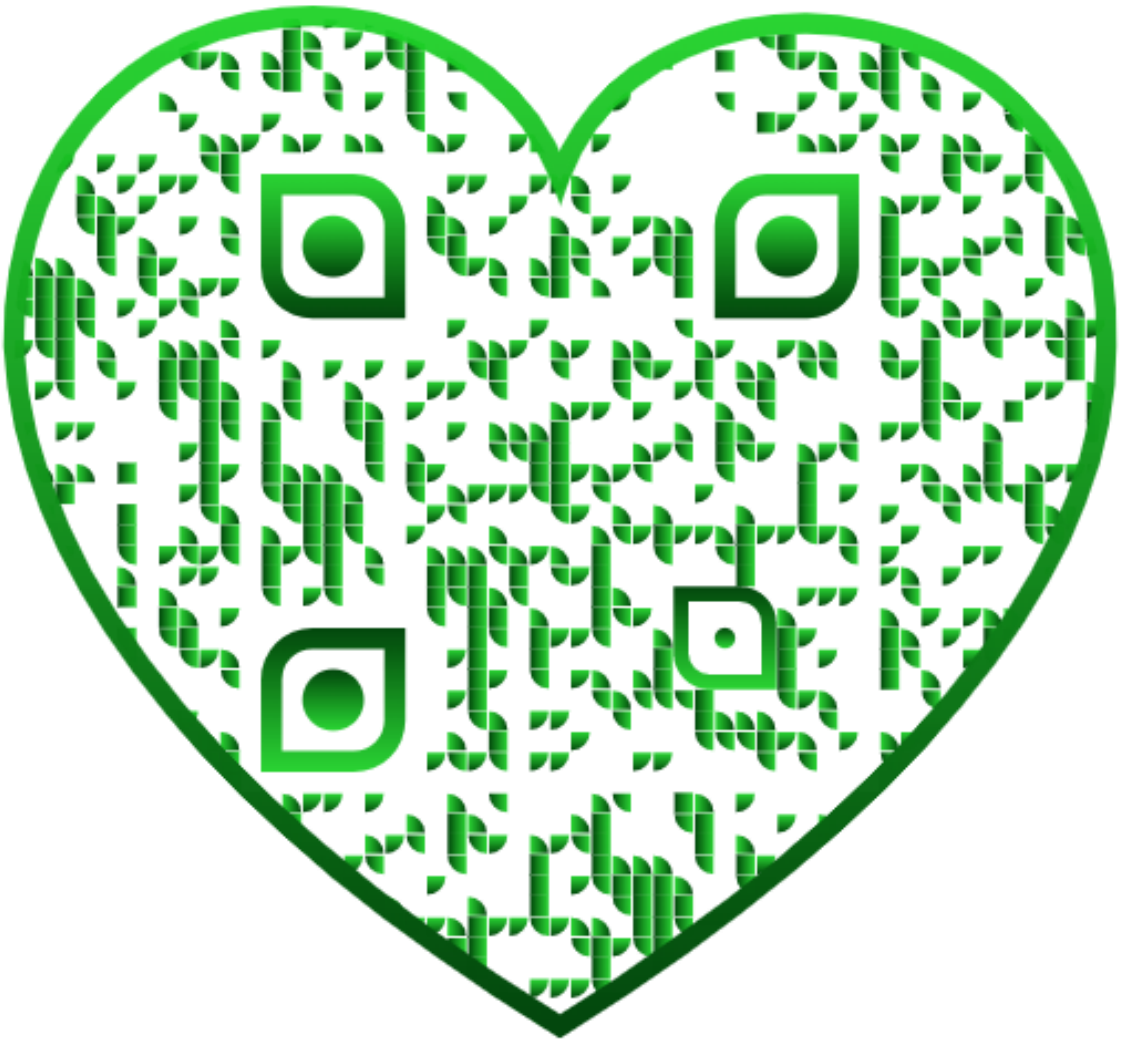
Pour des applications web puissantes en Python

Yohann Gabory

Préface de Nicolas Ferrari
Avec la contribution de Thomas Petillon



EYROLLES



Django avancé

Y. Gabory

Yohann Gabory est développeur web au sein d'Outscale et spécialiste de solutions Python/Django. Développeur Python depuis 7 ans, il a pratiqué d'autres framework tels que Ruby on Rails et Turbogears et a à son actif des références prestigieuses, telles que Libération et l'AFDAS. Il participe à des conférences Django et fut par ailleurs spécialisé en réseaux sociaux chez PilotSystems : protocole d'authentification Oauth, gestion d'identité OpenID, intégration Facebook, Twitter, Google Market Place... Il est aujourd'hui en charge du développement des interfaces de gestion du *cloud* Outscale.

Avec la contribution de Thomas Petillon

Django, framework Python MVC réputé pour son élégance et sa puissance, permet de développer des applications web de qualité professionnelle extrêmement riches et dynamiques en un temps record – pour peu que le développeur fasse l'effort de conception nécessaire et tire parti de son modèle de développement et de son écosystème.

Gagnez en expertise sur le framework de développement web le plus ambitieux !

Au fil de deux études de cas (un tracker et un agenda partagé) menées de bout en bout, cet ouvrage expose les fondamentaux de Django que sont les les vues, les templates, les formulaires et les modèles. Par sa présentation détaillée de l'architecture et des fonctionnalités de Django, cet ouvrage permet au développeur, aguerri comme débutant, d'atteindre une connaissance intime du framework.

Riche en exemples concrets et en astuces utiles, l'ouvrage distille également de nombreuses bonnes pratiques issues de l'expérience professionnelle de l'auteur. Il constitue pour le développeur Django le complément avancé indispensable à la documentation existante.

Au sommaire

Bien démarrer avec Django • Installer l'environnement de travail • Initialiser un projet • **Créer sa première application Django 1.5** • Un tracker • Méthode et framework • Mise en place du projet • Settings et configuration de la base de données (syncdb) et des applications • URL, application dynamique, objet ticket • Interfaces d'administration et client • **Des bases à la production** • Créer un agenda partagé • Organisation du projet : modèles et tables • **Migrations** • **Gestion de l'authentification** • Template login • Formulaire • Tests unitaires • **Création d'un événement** • ModelForm • Contraintes • Formulaire intelligent • Interface CRUD • **Templates pour l'affichage** • Intégration avec CSS et JavaScript • Organisation • Inclure un framework CSS • **Vues génériques** pour les fonctions d'agenda • Collection d'événements • Pagination • Détails d'un événement • **Revue de l'application et factorisation du code** • Vue Evenement_Detail : factorisation du formulaire et URL DRY • Créer, modifier, supprimer des événements • **Partage** • Gérer la liste d'utilisateurs connectés : modèles, invitations, vues • Relier les applications Calendrier et Agenda : gestion des participants et des statuts • **Bases de données** • Choisir sa base de données • **Django et le NoSQL** • ORM • Types de champs • Relations • Héritages • Méthodes des modèles • Querysets • Managers • Objet Q • **Traitement de l'information : les vues** • Workflow, middlewares, objets request et response, compilation du template, les context processors • **Vues génériques** • Vues classes *vs* vues fonctions • **Affichage de l'information : les templates** • Langage • Bonnes pratiques d'organisation • Tags et filtres • **Dialogue avec l'utilisateur : les formulaires** • Objet Form • Options des champs • **Django et le protocole web** • GET, POST et REST • Mise en place d'une API REST complète • **Contribs** • Sécurisation OAuth • Modules admin, content-types, sitemaps, syndication... • **Django avancé** • Surclasser l'application admin • Vues : générer modèles et URL à la volée • Utiliser Django en dehors de Django.

À qui s'adresse ce livre ?

– À tous les développeurs web (PHP, Java, Python, etc.) qui souhaitent recourir à un framework puissant pour des applications professionnelles ;

– Aux développeurs Django souhaitant aller plus loin dans leur maîtrise du framework.

Code éditeur : G13415
ISBN : 978-2-212-13415-5
Conception : Nord Compo

Django avancé

Pour des applications web puissantes en Python

Développement web et mobile

R. RIMELÉ. – **HTML5.**
N° 13638, 2e édition, 2013, 752 pages.

F. DRAILLARD. – **Premiers pas en CSS3 et HTML5.**
N° 13689, 2013, 472 pages.

R. GOETTER. – **CSS avancées.**
N° 13405, 2e édition, 2010, 385 pages.

R. GOETTER. – **CSS 2 : pratique du design web.**
N° 12461, 3e édition, 2009, 340 pages.

R. RIMELÉ. – **Mémento HTML5.**
N° 13420, 2012, 14 pages.

R. GOETTER. – **Mémento CSS 3.**
N° 13665, 2e édition, 2013, 14 pages.

É. DASPET et C. PIERRE DE GEYER. – **PHP 5 avancé.**
N° 13435, 6e édition, 2012, 870 pages.

C. PORTENEUVE. – **Bien développer pour le Web 2.0.**
N° 12391, 2e édition, 2008, 674 pages.

F. DAOUST, D. HAZAËL-MASSIEUX. – **Bonnes pratiques pour le Web mobile.**
N° 12828, 2011, 300 pages.

T. BAILLET. – **Créer son thème WordPress mobile.**
N° 13441, 2012, 128 pages.

É. SARRION. – **XHTML/CSS et JavaScript pour le web mobile.**
N° 12775, 2010, 274 pages.

É. SARRION. – **jQuery Mobile.**
N° 13388, 2012, 601 pages.

P. Y. CHATELIER. – **Objective-C pour le développeur avancé.**
N° 13686, 2e édition, 2013, 242 pages.

Design web et ergonomie

A. BOUCHER. – **Ergonomie web.**
N° 13215, 3e édition, 2011, 356 pages.

A. BOUCHER. – **Ergonomie web illustrée.**
N° 12695, 2010, 336 pages.

A. BOUCHER. – **Mémento Ergonomie web.**
N° 13735, 3e édition, 2013, 14 pages.

E. SLOÏM. – **Mémento Sites web.**
N° 12802, 3e édition, 2010, 14 pages.

O. ANDRIEU. – **Réussir son référencement Web.**
N° 13664, 2013, 552 pages.

F. MATTATIA. – **Traitement des données personnelles.**
N° 13594, 2013, 188 pages.

K. MCGRANE. – **Stratégie de contenu mobile.**
N° 13675, 2013, 172 pages.

C. SCHILLINGER. – **Intégration web : Les bonnes pratiques.**
N° 13370, 2012, 390 pages.

I. CANIVET et J-M. HARDY. – **La stratégie de contenu en pratique.**
N° 13510, 2012, 176 pages.

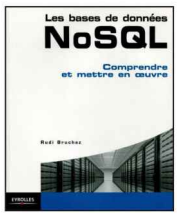
A. ALTINIER. – **Accessibilité web.**
N° 12889, 2012, 332 pages.

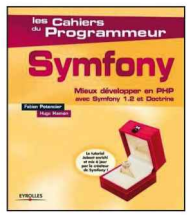
J. BUTTIGIEG. – **Mémento WordPress, programmation.**
N° 13596, 2012, 18 pages.

Autres ouvrages

F. MATTATIA. – **Traitement des données personnelles.**
N° 13594, 2013, 188 pages.

R. M. STALLMAN, S. WILLIAMS et C. MASUTTI. – **Richard Stallman et la révolution du logiciel libre.**
N° 13635, 2e édition, 2013, 338 pages.





Django avancé

Pour des applications web puissantes en Python

Yohann Gabory

Préface de Nicolas Ferrari
Avec la contribution de Thomas Petillon

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Remerciements à Thomas Petillon et Anne Bougnoux pour leurs contributions précieuses.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de copie, 20, rue des Grands Augustins, 75006 Paris.
© Groupe Eyrolles, 2013, ISBN : 978-2-212-13415-5

Préface

En quinze ans, les techniques de développement web ont beaucoup évolué. J'ai fait mes premiers pas sur Internet au milieu des années 1990 – après la récente introduction de PHP en 1994. Tout était prétexte à la création d'un site, à la fois pour nourrir la curiosité du développeur, mais aussi pour jouir d'une présence en ligne aux yeux de tous, et par delà les frontières. C'était une période de découvertes : on faisait tout au sein d'un même script, qu'il s'agisse d'interagir avec des bases de données, d'effectuer des traitements ou de générer un affichage HTML. Il n'existait alors pas de bibliothèques pour faciliter le développement, à l'exception de quelques frameworks assez lourds dont la prise en main était complexe si l'on était en phase d'apprentissage.

Ce n'est que plus tard, lors de la « bulle Internet » à partir de la fin des années 1990, qu'est arrivée une première vague de frameworks web formalisant, entre autres, un ensemble de patterns de développement permettant d'organiser le code de façon intéressante. Citons le fameux pattern MVC (Modèle-Vue-Contrôleur) qui propose de séparer clairement les données de leur traitement, et de leur affichage.

Fort de ces différents outils et ressources, le développement web s'est encore enrichi à partir de 2005, avec l'introduction de frameworks modernes construits sur des technologies récentes (Ruby on Rails, développé en Ruby) ou en pleine renaissance (Django, développé en Python). C'est d'ailleurs dans ce contexte que nous avons créé en 2006, avec mon ami Cyril, notre plate-forme d'hébergement alwaysdata.com, faite par et pour les développeurs, avec comme ligne directrice la prise en charge d'un maximum de technologies web.

En quelques années, Django a mûri : non seulement techniquement, puisque après une dizaine de versions antérieures, la version 1.5 devrait enfin voir le jour, mais aussi humainement, puisque sa communauté s'est étoffée avec l'organisation régulière de conférences en France et dans le monde. Django a désormais fait ses preuves dans le secteur du développement web, comme en témoignent de nombreuses applications : Disqus (la plus grosse plate-forme de gestion de commentaires), Instagram (rachetée

par Facebook), Pinterest, Mozilla Foundation, Libération, 20 minutes, Washington Post, Century 21, National Geographic – pour n'en citer que quelques-unes.

Cet ouvrage vient en renfort d'une documentation officielle en ligne déjà fort bien faite, tant pour les novices que pour les utilisateurs chevronnés de Django. Son auteur, Yohann, a également voulu démocratiser plus encore son framework de prédilection. Ainsi Yohann s'est affirmé en véritable « évangéliste » en mettant son expérience et ses compétences au service de ce livre. Passionné par Django depuis plus de six ans, sa participation à l'émancipation du framework l'a amené non seulement à réaliser de nombreux développements, mais aussi à donner des formations, des conférences et écrire des publications.

Son approche pour faire découvrir Django est très didactique : après une introduction sur la technologie et la démarche entreprise, il propose d'emblée d'entrer dans le vif du sujet en parcourant un spectre très large des fonctionnalités offertes par le framework, et ce, au fil du développement d'une application de A à Z. Vous disposez ainsi, avec ce livre, d'un des tutoriels les plus complets existant à ce jour sur Django. À l'issue de votre lecture, nul doute que vous aurez acquis les notions nécessaires au développement de votre prochaine application.

En espérant que cette lecture fasse naître des idées, je vous laisse entre les mains de Yohann et, pourquoi pas, sur un fond sonore de Jazz manouche...

Nicolas Ferrari

Développeur Django et co-fondateur d'alwaysdata.com

Table des matières

Avant-propos **1**
Pourquoi ce livre ? 1
À qui s'adresse le livre ? 1
Comment ce livre est-il organisé ? 2
Remerciements 3

CHAPITRE 1
Bien démarrer avec Django : installation et initialisation d'un projet **5**
Installer un bon environnement de travail 5
Choisir entre éditeur de texte et EDI 5
Installer Python 6
Installation sous Linux 6
Installation sous Mac OS X 6
Installation sous Windows 7
Installer pip et virtualenv 8
Installation sous Linux 8
Installation sous Mac OS X 9
Installation sous Windows 9
Installer un gestionnaire de versions 9
Les différents systèmes de gestion de versions 10
Installer un gestionnaire de versions : le choix Mercurial 11
Installer et tester Django 12
Initialiser un projet 13
Un projet, des applications 13
Les commandes d'initialisation d'un projet 14
Création d'un projet : startproject 14
Serveur web de développement : runserver 14
Conclusion 16

CHAPITRE 2

Créer sa première application : un tracker 19

De la méthode et un framework adaptable 19
Le développement par itérations 20
Un framework transparent et adaptable 21
De l'utilité d'un tracker 21
Organisation du travail 21
Gestion du temps 22
Gestion des jalons 22
Historique 22
Tracker et gestionnaire de versions : les outils d'une méthode « agile » 22
Initialisation du projet 23
Mise en place 23
Création d'une application : startapp 23
Vue d'ensemble du projet 23
Paramétrer les variables d'environnement (settings) 24
Mode de développement (DEBUG) 24
Configuration de la base de données 25
Internationalisation 26
Applications installées par défaut 27
Initialisation de la base de données : syncdb 28
Projet versus application 29
Configuration des applications installées 29
Les URL 30
Une première vue 32
L'objet request 34
L'objet HttpResponse 35
Créer une application dynamique 36
Le modèle de l'objet ticket 39
La classe Task 39
La commande python manage.py shell 41
Première itération : l'interface d'administration 43
Ajouter l'application aux settings 43
Ajouter l'application aux URL 44
Un premier test 44
Ajouter le modèle Task à l'administration 45
Une interface CRUD 46
Un premier aperçu 46
Comment sont créées les URL de l'administration ? 47
Les formulaires de l'administration 47
Modifier l'interface d'administration 47

La fonction __unicode__ 48
Aller plus loin dans la configuration de l'administration 49
Afficher les due-dates sur l'interface d'administration 49
Changer la couleur des champs due_date et schedule_date 50
Première conclusion 52
Seconde itération : les bases de l'interface client 53
L'application databrowse (contribs...) 53
Une première vue : liste 55
Introduction aux querysets 55
Présenter l'information : les templates 56
Les fichiers templates 57
Un raccourci : render_to_response 58
Troisième itération : l'interface utilisateur, aspects avancés 60
Mise en place des URL 60
Utiliser le dictionnaire GET 62
Les fichiers statiques 64
staticfiles 65
STATIC_URL et render_to_response 65
Une petite astuce 67
Conclusion 68

CHAPITRE 3
Des bases à la production : créer un agenda partagé 69
Fonctionnalités de l'application 70
Un environnement complet 70
Activer l'environnement virtuel 71
Installer les paquets de base 72
Créer un fichier requirements.txt 72
Configuration du projet 73
Plus d'efficacité avec south et local_settings de settings.py 76
Organisation du projet 79
Conception des modèles 79
La relation many-to-many 80
L'option through 80
Création des tables 80
Première migration avec south 81
Test des modèles et TDD 82
Les tests dans les docstrings 83
Conclusion 90

CHAPITRE 4
Gestion de l'authentification 91
Authentifier un utilisateur sur le site 91
Ajouter les URL login et logout 91
Créer le template login 93
Personnaliser le contenu 95
Une page pour chaque utilisateur 95
Une introduction aux vues génériques : TemplateView 96
Afficher les informations de l'utilisateur connecté 97
Autoriser ou interdire certaines parties du site 97
Découplage de l'application 98
Instancier un formulaire 100
Valider les données du formulaire 101
Enregistrer les données du formulaire 102
La fonction create_account finalisée 103
Les tests unitaires de l'authentification 103
La classe Client 104
Test des URL 104
Les tests des formulaires 106
Les tests d'échec 108
Derniers ajustements 109
Modifier les URL 109
Modifier les templates 112
Conclusion 114

CHAPITRE 5
Création d'un événement 115
Les ModelForm 115
Création du formulaire 116
Présenter l'objet Evenement à l'utilisateur 119
Ajouter des participants 121
Améliorations et finitions 125
Les contraintes 125
L'attribut unique pour la contrainte d'unicité 125
L'attribut unique_together 126
Rendre intelligent le formulaire 127
Présélectionner l'événement 127
Supprimer les choix inutiles 128
Supprimer un participant 129
N'afficher que les champs utiles 131

L'interface CRUD 132
Lister tous les événements 132
Supprimer un événement 134
Modifier un événement 134
Conclusion 135
Les URL : #agenda/personal_calendar/urls.py 135
Les vues : #agenda/personal_calendar/views.py 136
Les formulaires 137
Les modèles (sans les docstrings) :#agenda/models.py 137
Le template liste.html 138
Le template create.html 138
Le template details.html 139

CHAPITRE 6
Les templates pour l'affichage 141
Fonctionnement général : découplage et souplesse pour l'intégration avec CSS/JavaScript 141
Organisation des templates 142
Le fichier base.html 142
La directive extends 144
La directive include 146
Les styles CSS et les images 149
Servir les fichiers statiques durant le développement 149
Inclure un framework CSS dans Django 150
Intégrer du JavaScript 151
Calendrier JavaScript 151
Pliage et dépliage 153
Les requêtes Ajax 156
Conclusion 161

CHAPITRE 7
Les vues génériques pour les fonctions d'agenda 163
Les vues génériques depuis Django 1.3 163
Les bases des vues génériques 164
Afficher une collection d'événements 164
La pagination 167
La pagination dans les vues 168
Un peu de nettoyage 172
Afficher les détails d'un événement : DetailView 173
Conclusion 174

CHAPITRE 8
Une revue de l'application : factorisation du code 175
Utiliser les vues génériques dans l'ensemble de votre application 176
La vue Evenement_Detail 176
Factorisation du formulaire 178
Des URL DRY 180
Créer, modifier, supprimer des événements 182
La vue CreateView : création d'un nouvel événement 182
La vue UpdateView : mettre à jour un événement 183
La vue DeleteView : supprimer un événement 183
Conclusion 184

CHAPITRE 9
Les notions de partage 189
Le besoin : gérer sa propre liste d'utilisateurs connectés 189
Spécification des modèles 190
Définition des modèles 190
Les requêtes courantes 191
Gestion des invitations 192
Ajouter le champ email à la création d'un nouvel utilisateur 192
Le modèle Invitation 193
Le formulaire d'invitation 194
Envoi des e-mails avec Django 195
Créer le contact dans le carnet d'adresses de l'utilisateur 197
Rediriger vers la fiche contact et non l'invitation 198
Revue de code 200
Les vues 200
Le formulaire d'ajout d'un cercle 201
Le formulaire de modification d'un contact 201
Relier les applications Calendrier et Agenda 202
Présenter les utilisateurs du carnet d'adresses 202
Ajouter l'utilisateur créateur de l'événement à la liste des participants 203
La gestion des statuts 203
Empêcher la suppression de l'hôte 204
L'envoi d'un e-mail à l'ajout d'un participant 204
Laisser l'utilisateur indiquer son statut 205
Conclusion 211

CHAPITRE 10

Django et les bases de données 213

Tour d'horizon des bases de données relationnelles 213
MySQL : SGBD historique 213
PostgreSQL : robustesse 214
SQLite : légèreté 214
Comment choisir sa base de données ? 215
Tour d'horizon des bases de données NoSQL 215

L'ORM de Django : couche d'abstraction entre la BDD et les objets 217
Principe de fonctionnement 217
Une vision objet 217
Les principaux avantages : manipuler les objets de la BDD en Python 218
Les inconvénients : comment les contourner 218
Anatomie d'une queryset 218
Méfiez-vous des templates 220

Les différents types de champs 221
Les champs CharField 221
Les champs IntegerField 222
Les champs TextField 222
Les options remarquables 222
Options relatives aux formulaires 222
Options relatives à la base de données 222
Créez vos propres types de champs 223

Les relations 225
Les relations one-to-one 226
Cas d'utilisation 226
Implémentation 227
Queryset sur une relation one-to-one 227
Les foreign keys (clés étrangères) 228
Implémentation 228
Queryset sur une relation foreign key 228
Les relations many-to-many 230
Implémentation 230
Queryset sur une relation many-to-many 230
related_name 231
Manipulation des relations many-to-many 231

Les héritages de tables 232
Héritage abstract 232
Les classes abstraites et les applications réutilisables 233
Multitable 234

Proxy 234
Les méthodes des modèles 234
Quelques remarques : gérer les appels à la base de données 235
Les meta 236
Les échanges avec la base de données 236
L'ordre des résultats 236
Les contraintes d'unicité 237
Les permissions 237
Les méthodes prédéfinies 237
Les querysets 239
Description d'une queryset 239
Les relations dans les querysets 240
Le coût des querysets 241
Les managers 242
Le fonctionnement des managers 242
Faire ses propres managers 243
Ajouter de nouvelles méthodes 243
Modifier le fonctionnement du manager 244
Un exemple complet pour comprendre l'intérêt des managers 245
L'objet Q : encapsuler des paramètres pour exécuter des requêtes 247
Description de l'objet Q 247
Quand faut-il utiliser l'objet Q ? 248
Conclusion 249

CHAPITRE 11
Le traitement de l'information : les vues 251
Tour d'horizon des vues 251
Le workflow standard 252
L'objet request 252
La méthode 253
Les en-têtes 254
Les informations fournies par Django dans la requête 254
L'objet response 255
Les codes HTTP fournis par Django 255
Les autres codes HTTP 256
La compilation du template 256
Le principe de base de la compilation d'un template 256
render_to_response 257
L'objet RequestContext 258
render 259
Les context processors 259

Les vues et les objets de modèles 260
Les vues avancées 261
Organiser les vues d'un projet 261
Cas particulier : Ajax 262
Reconnaître une requête Ajax 262
Le contexte JSON 263
serializer 263
Les relations et JSON 263
Aller plus loin dans la sérialisation 264
Cas particulier CSV : format d'échange de données tabulaires 265
Renvoyer du CSV avec le moteur de templates Django 266
Cas particulier PDF 267
Un exemple simple 267
Avant et après la vue : les middlewares 268
Les vues génériques 269
Un exemple simple : TemplateView 269
ListView et DetailView 271
ListView et queryset 272
Les décorateurs et les vues génériques 273
Les mixins 274
MultipleObjectMixin 275
CreateView et DeleteView 275
CreateView 275
Le template CreateView 275
Les paramètres optionnels de CreateView 276
Les autres vues génériques 276
Les vues génériques basées sur les dates 276
Les vues génériques de dates de Django 277
Les vues-classes 278
Vues-classes vs vues-fonctions 278
Les verbes REST et les vues-classes 279
Les héritages de vues 279
Du bon usage des vues-classes 280
Utiliser les capacités d'agrégation de Django 284

CHAPITRE 12
L'affichage de l'information : les templates 285
Le langage de templates 285
Le template et le contexte 285
Afficher une variable 286
Afficher une série de variables 287

Branchements logiques 287
Les filtres 289
Revue de code 289
L'organisation des templates 289
Un dossier template pour le projet 290
Un dossier template par application 290
Les dossiers includes 291
Créer des template tags et des filtres 291
Les filtres : modifier une variable 292
Structure de fichier 292
Écriture d'un premier filtre 292
Les filtres avec argument 294
Les tags : agir au niveau du contexte 294
Les décorateurs 296
Ajouter des arguments aux tags 296
Conclusion 298

CHAPITRE 13
Dialogue avec l'utilisateur : les formulaires 299
L'objet Form 299
Un premier exemple : formulaire d'enregistrement à une newsletter 300
Traitement du formulaire 300
Affichage du formulaire 301
Les champs de formulaire 302
Les options remarquables d'un champ de formulaire 303
L'option required 303
La validation des champs de formulaire 304
Les options d'affichage 307
Les widgets 308
Modifier l'affichage HTML d'un formulaire 312
Conclusion 314

CHAPITRE 14
Django et le protocole web 315
Les verbes du Web 315
GET et POST 315
Les verbes REST 316
Un exemple concret : mise en place d'une API complète 317
Installation de Mezzanine 317
Description de OAuth2 318
Création de l'authentification OAuth2 318

Création de l'API ... 319
Sécuriser l'API ... 320
Authentification ... 321
Limiter l'accès aux ressources ... 322
Créer un client ... 322
slumber un client d'API REST ... 324
En bref ... 325
Une API REST en détail ... 326
Retrouver une ressource unique ... 326
Créer une nouvelle ressource ... 327
Modifier une ressource ... 328
Supprimer une ressource ... 330
Filtrer les ressources ... 330
Conclusion ... 333

CHAPITRE 15
Les contribs Django ... 335
Le module auth : authentification des utilisateurs ... 335
Installation ... 335
Choix du backend ... 336
RemoteUserBackend ... 336
L'objet User ... 337
Création d'utilisateur ... 338
Activer un utilisateur ... 339
Connecter un utilisateur ... 339
Déconnecter un utilisateur ... 342
Vérifier qu'un utilisateur est connecté ... 343
La gestion des utilisateurs connectés dans les templates ... 344
Les permissions ... 345
Les permissions d'un utilisateur ... 345
Les groupes ... 347
Créer ses propres permissions ... 349
En bref ... 349
Le module admin : gérer l'interface d'administration de Django ... 350
Cas d'utilisation du module admin ... 350
Le prototypage ... 350
L'interface d'administration à usage de maintenance ... 353
L'interface d'administration pour les usagers ... 354
L'interface d'administration par l'exemple ... 355
Présenter une liste des objets ... 355
Rechercher dans la liste d'objets ... 360

Les formulaires dans l'interface d'administration 362
Aller plus loin avec l'interface d'administration 366
Les méthodes spéciales des ModelAdmin 366
En bref 367
Les autres modules contribs 368
Les commentaires 368
Activer les commentaires pour votre projet 369
Utilisation des commentaires dans les templates 369
Prendre le contrôle des commentaires Django 370
En bref 371
Les content-types 371
Une application de tag utilisant les content-types 371
En bref 374
L'application flatpages : gérer les parties statiques de votre site 374
Installation 374
Fonctionnement 374
Le framework de message 375
Installation 375
Utilisation 375
L'application sitemaps : produire une carte de votre site 376
Installation 377
La classe Sitemap 377
Les raccourcis pour la création de sitemaps 378
En bref 379
Le framework de site : gérer plusieurs sites à la fois 379
Installation 379
Utilisation 379
Le framework de syndication : créer un flux RSS 381
Installation 381
Utilisation 381
Aller plus loin avec les flux RSS 382
En bref 383
Conclusion 383

CHAPITRE 16
Django avancé 385
Percer les secrets de l'interface d'administration 385
Les templates de l'administration 385
Révision des règles de surclassement de templates 386
Une application thème 387
Surclassement des templates d'administration 387

Surclasser change_list_result 387
En faire plus avec les vues : générer modèles et URL à la volée 388
Une application de prototypage 389
Les URL 389
Les vues 390
Les modèles 391
Les templates 393
Aller plus loin avec les templates 394
Utiliser Django en dehors de Django : déjouer l'anicroche 395

Index **397**

Avant-propos

Le Web a pris une place considérable dans notre quotidien et rend d'innombrables services. À mesure que les sites sont devenus plus riches et interactifs, les outils pour les créer ont évolué, faisant émerger des standards, des techniques et des architectures logicielles variés.

Le besoin s'est alors fait sentir de reformuler ce travail au sein d'ensembles cohérents permettant de manipuler des concepts abstraits, et avec des méthodes de travail adaptées aux nouveaux standards. C'est ainsi qu'est né le framework Django, offrant une boîte à outils pour créer des applications riches et modernes... « pour perfectionnistes pressés », comme le disent ses créateurs ! Nous vous invitons au travers de ce livre à en découvrir toute la puissance.

Pourquoi ce livre ?

Nombreux sont les sites français écrits avec le framework Django, et pas des moindres : Libération, 20minutes, Autolib, etc. Et pourtant, point de documentation en français à destination des développeurs web francophones. Ce livre vient combler cette lacune et expose tous les concepts fondamentaux à maîtriser, en les illustrant d'exemples et de cas pratiques.

À qui s'adresse le livre ?

Ce livre vient prolonger l'ouvrage *Apprendre la programmation web avec Python et Django* de Pierre Alexis et Hugues Bersini. Il suppose chez le lecteur, qu'il soit amateur éclairé ou développeur professionnel, une connaissance de Python. Il s'adresse

aussi bien au développeur souhaitant aborder un nouveau framework, qu'à celui ou celle qui ne connaît pas encore le développement web.

Quant au lecteur qui connaît et utilise déjà Django, il verra au fil d'exemples réels comment résoudre certains problèmes complexes auxquels il peut être confronté.

Comment ce livre est-il organisé ?

Ce livre est composé de trois parties qui suivent la progression du lecteur dans sa maîtrise de Django.

La **première partie** est composée de deux exemples d'apprentissage. Le premier explique les fondement de Django, son installation, sa prise en main et la création d'une première application, à savoir un **outil de suivi de bogues**. Le second permet très tôt de manipuler les concepts plus avancés de Django en créant un **agenda partagé**.

Dans la **deuxième partie** de ce livre, vous apprendrez à utiliser les différentes facettes de Django avec des exemples permettant de reproduire facilement les concepts que nous aborderons : gestion de la base de données, traitement de l'information et manipulation de formulaires seront au programme.

Enfin, dans la **troisième partie** de ce livre, vous découvrirez des méthodes bien plus avancées qui vous permettront de tirer parti de Django de manière parfois inattendue : après une revue des applications contribs vous manipulerez et créerez des API REST avec une authentification OAuth2. Nous dévoilerons ensuite quelques secrets du framework Django tirés de notre expérience professionnelle, avec entre autres un générateur de classes pour le prototypage d'applications.

Remerciements

Avant toute chose, je souhaite remercier très chaleureusement Karine Joly, mon éditrice, qui a su me soutenir et me guider tout au long de l'écriture de ce livre et toute l'équipe des éditions Eyrolles : Anne Bougnoux, Sophie Hincelin, Anne-Lise Banéath, Laurène Gibaud, Gaël Thomas et Muriel Shan Sei Fan, pour son soutien de tous les instants. Je tiens également à remercier Thomas Petillon, qui a profondément contribué à cet ouvrage par sa relecture technique et ses conseils avisés – terriblement précieux pour moi.

Je souhaite également remercier mon ami de longue date, Rodolphe Quiédeville, pour m'avoir mis entre les mains un premier CD-Rom de Debian, et m'avoir ouvert les portes du logiciel libre. Ce livre n'aurait tout simplement pas existé sans lui.

Merci à toute l'équipe de PilotSystems pour m'avoir soutenu lors de la rédaction de ce livre : David Sapiro, Gaël le Mignot, Léo Bernard, Bruno Dupuis, Eve Marie Thivolle.

Merci à Christelle, Esteban et Louise. Pour être là.

Merci enfin à tous les autres, qui se reconnaîtront.

1

Bien démarrer avec Django : installation et initialisation d'un projet

Après l'installation des outils qui nous serviront tout au long de l'ouvrage, nous initialiserons un premier projet.

Installer un bon environnement de travail

L'environnement de travail dans lequel vous allez évoluer et les outils utilisés sont très importants pour la qualité future de votre développement.

Choisir entre éditeur de texte et EDI

En tant que développeur, vous avez sans doute un outil de prédilection que vous ne changeriez pour rien au monde. Certains travaillent avec un « simple » éditeur de texte, comme Notepad++ (sous Windows), Gedit ou Geany (sous Linux), ou encore TexMate (sous Mac OS X). D'autres optent pour un environnement de développement intégré (EDI) comme Eclipse, surtout s'ils viennent du monde Java. Mais peut-être préférez-vous ces éditeurs d'un autre âge que sont Emacs ou Vim, certes antédiluviens mais terriblement puissants ? Dans tous les cas, Django s'intégrera parfaitement à la solution que vous aurez retenue.

Installer Python

Python est le langage de programmation sur lequel est basé Django, son installation est nécessaire (et nous conseillons son apprentissage).

Références

- G. Swinnen, *Apprendre à programmer avec Python*, Eyrolles.
- P. Alexis et H. Bersini, *Apprendre la programmation web avec Python et Django*, Eyrolles.

Version Pourquoi Python 2.7 ?

C'est en décembre 2008 que la version 3.0 du langage Python a vu le jour. Elle est incompatible avec les précédentes versions, numérotées 2.*x*. Or c'est la version 2.7 du langage qui peut être vue comme celle stable à ce jour.
L'équipe de développement de Django est en train de mener un énorme travail de refonte afin que le framework devienne compatible avec la série 3.x. Ce travail est aujourd'hui bien avancé, et un support bêta de Python 3 a vu le jour dans la toute récente version 1.5 de Django. Il faudra attendre la version 1.6 du framework pour que Django soit pleinement compatible avec Python 3.

Installation sous Linux

L'installation de Django sur un système GNU/Linux est très simple. Qu'il soit au format RPM (CentOS, RedHat, Fedora…) ou sous forme de paquet `.deb` (Ubuntu, Debian…), votre système fournit tout le nécessaire pour une installation simplifiée. Vous n'avez donc rien à faire pour le moment. En effet, c'est votre gestionnaire de paquets qui se chargera d'installer pour vous Django et ses dépendances, notamment Python, s'il n'est pas déjà installé sur votre machine.

Installation sous Mac OS X

Sous ses dehors de système d'exploitation du XXI^{e} siècle, Mac OS X est en réalité basé sur un très vieux système : Unix – plus précisément BSD, un Unix libre. C'est d'ailleurs la raison du « X » de Mac OS X. Inventé dans les années 1970, ce système est particulièrement bien adapté à un environnement de développement. Ainsi, vous n'aurez pas de difficultés majeures à installer tout ce dont vous aurez besoin pour faire fonctionner Python et Django.

Xcode

Afin de bénéficier du meilleur environnement qui soit, nous vous conseillons fortement d'installer Xcode. C'est un ensemble de logiciels gratuits, à télécharger sur l'App Store, qui permet de transformer votre ordinateur en plate-forme de développement.

Si votre choix n'est pas encore fixé sur un éditeur de texte en particulier, vous pourrez utiliser celui livré avec Xcode.

Python

Python étant installé par défaut sous Mac OS X, vous n'avez rien de plus à faire pour cette première partie.

Installation sous Windows

Windows n'étant pas le plus accueillant des systèmes d'exploitation pour le développement informatique, quelques étapes de préparation sont nécessaires avant d'avoir un environnement fonctionnel.

Cygwin

Tout d'abord, installez Cygwin (http://www.cygwin.com) qui va créer une base accueillante pour Python et Django, notamment par le biais d'une console. Il vous sera demandé les outils que vous souhaitez installer. Voici ceux dont vous aurez l'utilité :

- Python ;
- Mercurial (un gestionnaire de versions, voir plus loin) ;
- GCC et GCC+ (pour compiler en langage machine certains modules qui seront utiles tout au long de ce livre) ;
- Wget (petit utilitaire fort pratique qui installe les outils de base de Python).

À la fin de l'installation, n'oubliez pas de créer une icône sur le Bureau, comme cela vous est proposé, afin de retrouver plus rapidement la console ou le terminal dont vous allez faire usage régulièrement. Double-cliquez sur cette nouvelle icône. Un écran noir apparaît : il s'agit de la console.

Les outils setuptools

Depuis cette console, vous allez installer l'ensemble d'outils appelé `setuptools`, qui sert à installer des programmes Python sur votre système.

Rendez-vous à l'adresse http://pypi.python.org/pypi/setuptools. Dans la liste de téléchargement, copiez le lien qui correspond à votre installation de Python. À l'heure où nous écrivons ces lignes, il s'agit de la version 2.7.

Collez ce lien dans votre console, précédé de `wget` – cela va télécharger sur votre ordinateur une copie des setuptools qui vous seront utiles.

```
$ wget http://pypi.python.org/packages/2.7/s/setuptools/setuptools-0.6c11-
py2.7.egg#md5=fe1f997bc722265116870bc7919059ea
```

Ensuite, toujours dans la console, saisissez le nom du fichier téléchargé, ici `setuptools-0.6c11-py2.7.egg`, précédé de `sh` :

```
sh setuptools-0.6c11-py2.7.egg
```

Vous avez désormais à votre disposition une nouvelle commande nommée `easy_install`, qui facilite l'installation des paquets Python (ou *packages*). Ces derniers sont l'équivalent de programmes ou de bibliothèques tierces qui vous seront utiles tout au long de ce livre, et probablement par la suite.

Installer pip et virtualenv

Au cours de ce livre et dans votre parcours de développeur Django, vous allez réaliser régulièrement de nouveaux projets. Seulement, ces derniers n'ont pas vocation à rester sur votre ordinateur personnel mais à être placés sur des ordinateurs distants : les serveurs. Il est probable que votre machine de développement ne soit pas identique à la machine de production, et qu'elles n'auront pas le même système d'exploitation. Il vous faudra donc un système qui permette d'isoler vos projets afin de les dupliquer facilement d'une machine à l'autre. Deux outils vous seront nécessaires.

- `pip`, programme qui permet d'installer facilement des paquets Python, quel que soit votre système. Il fournit la liste des paquets déjà installés, les met à jour et les supprime éventuellement. Il sait également installer automatiquement une liste prédéfinie de paquets.
- `virtualenv`, un gestionnaire et créateur d'environnements virtuels. Il crée des espaces de travail isolés du reste de votre machine. Ainsi, lorsque vous installez un logiciel Python dans un environnement virtuel, il n'est disponible que dans cet environnement et non sur l'ensemble de votre système.

Installation sous Linux

L'installation des deux outils se fait très simplement via votre gestionnaire de paquets. Sur un système à base de paquets `deb` :

```
$ sudo apt-get install python-pip python-virtualenv
```

et sur un système à base de RPM :

```
$ sudo yum install python-pip python-virtualenv
```

Il est très probable que Django soit déjà dans les dépôts de votre distribution. Tapez simplement, sur un système à base de `deb` :

```
$ sudo apt-get install python-django
```

ou sur un système à base de RPM :

```
$ sudo yum install python-django
```

Installation sous Mac OS X

Ouvrez simplement un terminal – ce programme est déjà installé dans le répertoire `Application/Accessoires` – et tapez :

```
$ sudo easy_install pip
```

puis :

```
$ sudo pip install virtualenv
```

Puisque vous avez utilisé `sudo`, vous serez invité à entrer votre mot de passe de super-utilisateur. Ainsi, vous pourrez installer ces deux programmes sur l'ensemble de votre système, vous permettant d'en bénéficier à tout moment.

Installation sous Windows

Vous devez commencer par installer `pip`. Dans votre console Cygwin, tapez :

```
$ easy_install pip
```

Puis `virtualenv` s'installe directement avec votre nouveau logiciel :

```
pip install virtualenv
```

Installer un gestionnaire de versions

Maintenant, il vous faut installer un outil de suivi de code : un gestionnaire de versions.

MÉTHODE Pourquoi utiliser un gestionnaire de versions ?

Si vous avez déjà fait du développement informatique, vous connaissez l'intérêt d'un gestionnaire de versions. Si ce n'est pas le cas, nous vous proposons de suivre un petit exemple. Imaginez que vous souhaitiez élaborer une nouvelle recette de cuisine. Vous voulez qu'elle soit parfaite car vous participez au concours de la meilleure recette d'omelette du monde. Vous allez donc devoir la tester et la retester afin de l'améliorer par étapes successives. Pour commencer, vous partez sur une recette assez simple. Vous notez cette recette sur une fiche de cuisine et la testez sur vos convives. Même si cette omelette est réussie, vous trouvez que les lardons manquent de cuisson. Vous allez donc modifier votre mise en œuvre et ajouter les lardons à la préparation, au même moment que les œufs.

Ingrédients : *4 œufs, 150 grammes de lardons, beurre et fromage râpé.*
Mise en œuvre : *Battre les œufs longuement dans une jatte. Beurrer une poêle à fond plat. Faire chauffer le beurre sans le laisser brunir et ajouter les œufs. Lorsque ces derniers commencent à prendre, verser les lardons. En fin de cuisson, ajouter le fromage râpé. Une fois celui-ci fondu, plier l'omelette et servir avec une salade de mesclun.*

Vous faites donc une seconde fiche. Et vous testez de nouveau la recette. Le résultat vous plaît, mais vous aimeriez tester avec des lamelles de pommes de terre. Vous ajoutez donc les pommes de terre à vos ingrédients, modifiez la mise en œuvre et retestez la recette.
Vous allez donc rapidement devoir classer vos fiches. Peut-être faut-il les numéroter ? Mais dans quel ordre ? Chronologique ? Et que faire alors des essais ? Des fiches d'une autre couleur ? Mais comment s'y retrouver quand on réalise plusieurs essais en modifiant la même recette de base ? Il vous faut analyser posément vos besoins.

- Conserver toutes les **versions** de votre recette.
- Garder en mémoire les **différences** entre les recettes.
- Pouvoir **annuler** certains changements apportés.
- Marquer différentes « **branches** » de la même recette de départ.

C'est exactement ce que fait un gestionnaire de versions. Ce logiciel va « suivre » l'évolution de certains fichiers présents sur votre ordinateur, en mémoriser tous les changements pour vous permettre de revenir quand vous le souhaitez à une version antérieure. Vous pourrez également ôter les recettes finalisées, créer de nouvelles versions de la même recette et les fusionner quand les tests sont concluants.
Si cela vous semble un peu perturbant, rassurez-vous ! Dès que vous commencerez à vous en servir, vous ne pourrez plus vous en passer.

Les différents systèmes de gestion de versions

Les gestionnaires de versions existent depuis la nuit des temps informatiques – le premier fut SCCS en 1972. Depuis, leur nombre a augmenté de façon exponentielle. Aujourd'hui, on peut les séparer en deux groupes distincts : les gestionnaires centralisés (CVS ou SVN) et les gestionnaires distribués (Mercurial ou Git).

Les premiers offrent une architecture où le dépôt est unique et installé sur une seule machine. A contrario, les seconds proposent une architecture où chaque participant possède intégralement le dépôt et tout son historique. Dans la pratique, ces différentes architectures n'ont d'importance que lors d'un travail à plusieurs personnes. Dans le cas d'un seul développeur, vous ne verrez pas de différences notoires entre ces deux architectures.

Installer un gestionnaire de versions : le choix Mercurial

Tout d'abord, vous devez choisir entre gestionnaire de versions distribué ou centralisé. Comme aujourd'hui la tendance va plutôt vers une architecture décentralisée, nous avons choisi de vous proposer un outil présentant cette architecture. Les deux gestionnaires de versions distribués disponibles actuellement sont Mercurial et Git. Même s'ils sont d'aussi bonne qualité l'un que l'autre, nous avons opté pour Mercurial, que nous trouvons un peu plus simple d'utilisation. Tous les exemples présentés dans ce livre utiliseront donc ce gestionnaire de versions. Sachez que ses commandes sont très proches de celles de Git ; si un jour vous devez changer d'outil, vous n'aurez ainsi pas de difficultés majeures à vous adapter.

POUR EN SAVOIR PLUS Git

📖 R. Hertzog, P. Habouzit, *Mémento Git à 100 %*, Eyrolles, 2012.

L'installation de Mercurial dépend bien entendu de votre système d'exploitation.

- Sous Windows, vous l'avez déjà installé lors de la configuration de Cygwin.
- Sous Mac OS X, rendez-vous sur le site de Mercurial (http://mercurial.selenic.com). Téléchargez, puis installez le binaire.
- Sous Linux, Mercurial est déjà présent dans vos dépôts. Vous pouvez donc l'installer simplement en suivant la méthode habituelle. Par exemple, sous Debian :

```
apt-get install mercurial
```

Pour vérifier que l'installation s'est correctement déroulée, ouvrez un terminal et tapez :

```
$ hg
```

Vous devriez voir apparaître ce message, ce qui vous indique que l'installation s'est bien déroulée :

```
Mercurial Distributed SCM

basic commands:

add        add the specified files on the next commit
annotate   show changeset information by line for each file
clone      make a copy of an existing repository
commit     commit the specified files or all outstanding changes
diff       diff repository (or selected files)
```

```
export      dump the header and diffs for one or more changesets
forget      forget the specified files on the next commit
init        create a new repository in the given directory
log         show revision history of entire repository or files
merge       merge working directory with another revision
pull        pull changes from the specified source
push        push changes to the specified destination
remove      remove the specified files on the next commit
serve       start stand-alone webserver
status      show changed files in the working directory
summary     summarize working directory state
update      update working directory (or switch revisions)

use "hg help" for the full list of commands or "hg -v" for details
```

Installer et tester Django

Vous venez d'installer un ensemble d'outils fort intéressants mais, pour le moment, il n'y a pas de Django à l'horizon. Il est donc temps de tester votre installation.

Commencez par sélectionner un espace de travail dans lequel vous rangerez vos différents projets. C'est à vous de décider ce qui vous convient le mieux : sur votre Bureau, dans un répertoire personnel... Pour nos exemples, nous avons choisi de créer notre dossier de travail à la racine de notre répertoire personnel et de le nommer `DEV`.

Via le terminal, rendez-vous donc dans votre dossier de développement. Créez-y votre premier environnement virtuel :

```
$ virtualenv environnement_de_test
New python executable in environnement_de_test/bin/python
Installing setuptools.............done.
Installing pip...............done.
```

Activez ensuite votre nouvel environnement virtuel :

```
$ source environnement_de_test/bin/activate
(environnement_de_test)$
```

Entre parenthèses, avant votre invite de commande, vous voyez le nom de l'environnement virtuel actuellement actif ; cela vous évitera de vous tromper d'environnement au cours de votre travail.

Quand vous souhaiterez quitter cet environnement virtuel ou en changer, tapez simplement :

```
(environnement_de_test)$ deactivate
$
```

Ne le faites pas pour le moment car vous allez enfin installer Django avec la commande suivante :

```
(environnement_de_test)$ pip install django
Downloading/unpacking django
  Downloading Django-1.4.3.tar.gz (7.7Mb): 7.7Mb downloaded
  Running setup.py egg_info for package django

Installing collected packages: django
  Running setup.py install for django
    changing mode of build/scripts-2.7/django-admin.py from 644 to 755

    changing mode of /Users/user/Dev/environnement_de_test/bin/django-admin.py
to 755
Successfully installed django
Cleaning up...
(environnement_de_test)$
```

Django est maintenant installé dans votre environnement virtuel. Pour vérifier votre installation, tapez :

```
$ django-admin.py --version
1.4.3
```

`django-admin.py` est un exécutable fourni avec Django qui va vous servir à créer de nouveaux projets.

Initialiser un projet

Maintenant que vous possédez l'outillage du parfait développeur Django, vous pouvez créer votre premier projet.

Un projet, des applications

Django est construit sur une approche très modulaire ; il implémente avec efficacité les notions de découplage et de réutilisabilité. Pour y parvenir, il découpe chaque projet en un ensemble de plus petits paquets nommés « apps ».

Lorsque vous souhaitez créer une application web, vous réalisez donc un projet dans lequel résideront différentes applications. Ces dernières pourront aisément être déplacées d'un projet à l'autre, vous permettant, dans l'organisation même de vos répertoires, d'être modulaire et DRY (*Don't Repeat Yourself* ou « on ne duplique pas de code »).

Django fournit un ensemble de commandes qui faciliteront la gestion de votre projet tout au long de sa vie. Nous verrons même, dans les chapitres suivants, comment créer soi-même des commandes de ce type. Pour le moment, voyons d'abord celles qui vous seront utiles immédiatement.

Les commandes d'initialisation d'un projet

Pour ce premier exemple, utilisez l'environnement virtuel que vous venez de réaliser. Pour mémoire :

```
$ source environnement_de_test/bin/activate
(environnment_de_test)$
```

Avant de tester les différents extraits de code, vérifiez que vous êtes bien dans le bon environnement de test.

Création d'un projet : startproject

La première commande que vous allez utiliser est :

```
[user@local]$ django-admin startproject premiertest
```

Cette fonction réalise l'arborescence nécessaire à votre projet. Vous pouvez constater qu'un dossier `premiertest` vient d'être créé.

Serveur web de développement : runserver

Pour le moment, nous pouvons d'ores et déjà vérifier que tout est fonctionnel, en lançant le serveur de développement.

Rendez-vous tout d'abord dans le répertoire de votre projet :

```
[user@local]$ cd premiertest
[user@local]$ python manage.py runserver
Validating models...

0 errors found
Django version 1.4.3, using settings 'premiertest.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Le retour du terminal vous invite à vous rendre à l'adresse `http://127.0.0.1:8000`, qui correspond à votre propre machine sur le port 8000. En effet, Django vient de lancer pour vous un serveur web rudimentaire, particulièrement bien adapté pour le déve-

loppement, grâce auquel vous testerez facilement votre application sur votre machine. Entre autres fonctionnalités, ce serveur de développement « ausculte » en permanence votre projet et, dès qu'un fichier est modifié, il se recharge automatiquement. Ainsi, vous pourrez apprécier immédiatement un changement dans votre code, simplement en actualisant votre navigateur.

ATTENTION Pas d'utilisation en production

Ce serveur web ne devra jamais être employé en production : il n'est pas conçu pour accepter plus d'une connexion à la fois, prohibant son utilisation dans quelques autres cas que celui des tests ou du développement.

Sachez également que vous pouvez vous rendre à l'adresse `http://localhost:8000`, qui est un alias de `http://127.0.0.1:8000`. Si, pour une raison ou pour une autre, le port 8000 n'est pas disponible, voici le message d'erreur qui doit s'afficher :

```
Validating models...
0 errors found
Django version 1.3, using settings 'premiertest.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
Error: That port is already in use.
```

Il vous suffit alors de lancer la commande, suivie du numéro de port que vous souhaitez utiliser, par exemple 5000 :

```
$ python manage.py runserver 5000
```

Il existe d'autres options disponibles pour le serveur de développement de Django. Pour en avoir un aperçu, tapez :

```
python manage.py help runserver
```

Une fois votre navigateur web favori ouvert sur la page `http://localhost:8000`, vous verrez s'afficher l'écran suivant. Félicitations, l'installation de Django s'est parfaitement déroulée !

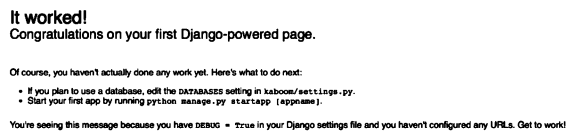

It worked!
Congratulations on your first Django-powered page.
Of course, you haven't actually done any work yet. Here's what to do next:
• If you plan to use a database, edit the DATABASES setting in kaboom/settings.py.
• Start your first app by running python manage.py startapp [appname].
You're seeing this message because you have DEBUG = True in your Django settings file and you haven't configured any URLs. Get to work!

Figure 1–1 Écran d'accueil de Django

Cette page vous invite à indiquer les paramètres de votre base de données dans le fichier `settings.py` et à créer votre première application.

Nous aimerions attirer votre attention sur un point intéressant. Revenez sur le terminal d'où vous avez lancé la commande `python manage.py runserver`. Vous devriez voir s'afficher une ligne de ce type :

```
[27/Jul/2011 12:44:10] "GET / HTTP/1.1" 200 2053
```

Il s'agit tout simplement de la requête que Django vient de vous servir, décomposée comme suit :

- la date et l'heure ;
- le verbe utilisé ;
- l'adresse demandée (ici `/`, la racine de votre site) ;
- le protocole utilisé ;
- la taille (en octets) de la réponse.

Cette fonctionnalité va vous être très précieuse dès que vous souhaiterez déboguer votre application. C'est un élément qui vous sera utile tout au long de ce livre, et même après !

Conclusion

Vous avez donc maintenant une installation fonctionnelle, qui est reproductible très simplement. Dès que vous souhaiterez réaliser un nouveau projet, vous n'aurez qu'à suivre ces quelques étapes :

1. Créer un nouvel environnement virtuel et l'activer :

```
$ virtualenv votre_nouvel_environnement
$ source votre_nouvel_environnement/bin/activate
```

2 Installer Django :

```
(votre_nouvel_environnement)$ pip install django
```

3 Initialiser un nouveau projet :

```
(votre_nouvel_environnement)$ django-admin nouveau-projet
```

Le projet est la pièce indispensable au bon fonctionnement d'une application web réalisée à l'aide du framework Django. Les différentes fonctionnalités résident pourtant dans des composants différents : les applications. Dans le prochain chapitre, en même temps que vous créerez votre premier outil, vous apprendrez à configurer un projet et à lui ajouter de nouvelles fonctionnalités à l'aide de ces nouveaux composants.

2

Créer sa première application : un tracker

Dans ce chapitre, vous allez créer votre première application. Ce faisant, vous découvrirez progressivement certaines des fonctionnalités majeures de Django. Loin des « wiki en vingt minutes » et autres « mon premier blog en Django », ce premier projet vous accompagnera tout au long du livre et, nous l'espérons, bien au-delà. Vous allez créer un tracker, outil qui tracera l'avancement de votre projet en vous donnant une vision claire de votre travail et du temps que vous y consacrez.

De la méthode et un framework adaptable

Nous n'allons pas créer d'emblée toutes les fonctionnalités de notre projet, de la base de données vers l'interface graphique, comme cela se faisait il y a encore quelques années avec le mode de développement par cahier des charges. Cette manière de procéder est à éviter à bien des égards. Entre autres, vous risquez les deux écueils suivants.

- **Oublier une fonctionnalité majeure.** Au beau milieu de votre projet, vous vous apercevez que vous avez oublié un élément primordial. Seulement, vous avez déjà créé la base de données et les fonctions principales du cœur de votre application ; modifier toutes ces fonctions est un tel travail que vous n'avez plus qu'à tout reprendre depuis le début ou, tout simplement, abandonner le projet.

- **Coder une fonctionnalité inutile.** Bien que ce cas paraisse moins grave que le précédent, il se produit plus souvent que l'on ne le pense. En imaginant l'application, vous croyez que telle ou telle fonctionnalité sera indispensable ou très utile. Vous codez donc votre application autour de cette pièce maîtresse. En réalisant les premiers tests fonctionnels, vous vous apercevez que cette fonction n'est finalement pas utile, voire qu'elle gêne le reste du projet. C'est bien souvent dans les dernières étapes, lorsque l'on dessine l'interface web de l'application, que les problématiques de ce genre se font jour. Il faut alors repenser entièrement le programme, ce qui peut bien souvent mener à l'abandon du projet

Une autre raison de rejeter le développement par « cahier des charges » est sans doute d'ordre plus personnel. En effet, lorsque l'on est seul sur son projet, avec pour seul appui sa motivation, devoir s'atteler à une tâche parfois de plusieurs mois avant de récolter les fruits de son travail est très vite décourageant.

Pour éviter ces écueils (et bien d'autres), nous vous proposons de découvrir dès votre premier projet une autre méthode : **le développement par itérations**.

Le développement par itérations

Le développement par itérations consiste à créer un programme par petites parties indépendantes les unes des autres. Si, par exemple, vous devez gérer une liste d'utilisateurs, vous commencerez par créer un objet utilisateur, puis l'interface nécessaire pour le manipuler. Démuni de fonctionnalités, cet embryon d'application n'en sera pas moins pleinement fonctionnel ; vous pourrez le manipuler, « jouer avec », le tester dans différentes situations et détecter très rapidement les éventuelles erreurs de conception. À ce niveau de la vie de l'application, il n'est évidemment pas question de régler tous les détails graphiques, tous les cas particuliers. Il s'agit seulement d'en dessiner les grandes lignes et de pouvoir manipuler l'objet pour découvrir très tôt ses failles et ses qualités. Une fois que ce bloc de fonctionnalités vous semblera satisfaisant, vous ajouterez alors de nouvelles briques à l'ensemble en procédant de la même manière. Ainsi, brique après brique, vous verrez votre application grandir, mûrir, évoluer avec, à chaque fois, la possibilité de modifier, transformer et adapter votre application à de nouveaux cas.

L'un des avantages principaux de développer une application web à l'aide d'un framework, et c'est particulièrement vrai avec Django, est d'obtenir rapidement des résultats tangibles, ce qui est évidemment bienvenu dans le cadre d'un développement par itérations. Dès cette première application, vous allez vous rendre compte à quel point Django vous offre la possibilité de modéliser rapidement les premières briques en quelques commandes seulement.

Un framework transparent et adaptable

Très souvent, cette rapidité de développement souffre d'un revers de taille. En effet, lorsque vous souhaitez sortir de ce qui est prévu par le framework, les choses commencent à se gâter. Mais, après plusieurs années de développement avec Django, nous vous assurons qu'il n'en est rien.

Cela tient en définitive à deux choses.

- Tout d'abord, Django repose sur le langage de programmation Python. La nature interprétée et dynamique de ce dernier rend son utilisation très souple et polymorphique : il est par conséquent très simple de modifier la façon dont fonctionne ici une fonction, là une classe.
- La seconde raison tient à une volonté précise des créateurs de Django qui, le 1er mai 2006, ont pris la décision de supprimer la « magie » de ce framework. Cela a principalement consisté à enlever tous les éléments qui cachaient aux développeurs le fonctionnement interne de Django. Désormais, ils peuvent modifier entièrement son comportement pour un besoin précis et l'adapter à volonté. À la fin de ce livre, dans la partie consacrée aux techniques avancées avec Django, nous aborderons plusieurs exemples de modifications de ce type.

Nous allons vous guider pas à pas dans le fonctionnement interne de Django. L'idée n'est pas de vous assommer dès le début avec des concepts complexes, mais de reprendre son principe de fonctionnement grâce à un exercice didactique et d'en retirer toute la magie (ou au moins une partie).

De l'utilité d'un tracker

Comme premier exercice, nous vous proposons de créer votre premier outil : un *tracker*.

Organisation du travail

Dans sa forme la plus basique, un tracker est une sorte de liste de choses à faire qui aide à maintenir un cap dans l'évolution d'un projet. Élément indispensable du travail en équipe, il en assure la cohésion et indique à tous qui fait quoi et sur quelle partie du code.

Même lors d'un travail solitaire, un tracker vous rendra de fiers services. Lorsque vous reviendrez à votre projet après quelques jours d'inactivité, vous saurez, grâce à lui, quelles tâches il vous reste encore à accomplir. Vous en garderez ainsi une vision claire et maintiendrez son évolution avec sérénité.

Gestion du temps

Un tracker vous aidera également à gérer votre charge de travail en fonction de vos disponibilités, à planifier les futures évolutions de votre projet et à estimer le temps passé sur telle ou telle fonctionnalité. Cette capacité à gérer votre projet dans le temps garantit une évolution stable et vous fournit les atouts indispensables à sa réussite.

Gestion des jalons

Un autre point fort d'un tracker réside aussi dans l'utilisation intelligente de jalons qui marqueront les différentes étapes de votre projet. Un jalon se définit comme une liste des choses qu'il reste à accomplir avant qu'une partie ou une fonctionnalité du programme soit opérationnelle. C'est particulièrement utile dans un mode de développement par itérations, où chaque jalon correspond à une itération précise. Un tracker vous permettra donc de mesurer et de suivre l'évolution de votre projet, jalon après jalon.

Historique

Enfin, vous serez en mesure de garder une trace de votre travail et du temps nécessaire à son accomplissement. Comme vous enregistrerez le temps passé sur chacune de vos tâches, et donc le temps global passé sur un jalon, vous pourrez y revenir pour estimer la somme des efforts qu'il vous aura fallu fournir pour mener à bien votre tâche.

Avec le temps, vous estimerez de plus en plus finement le temps nécessaire à la réalisation d'une tâche. Vous pourrez la planifier, l'organiser et donc rester maître du temps, notion essentielle en programmation (mais pas seulement !).

Tracker et gestionnaire de versions : les outils d'une méthode « agile »

Comme nous l'avons vu dans le chapitre précédent, le gestionnaire de versions est lui aussi indispensable à la bonne conduite d'un projet. Nous ne pouvons que vous en conseiller très fortement l'usage, quelle que soit la taille de votre projet, même s'il ne s'agit que d'un petit script dans un coin de votre disque dur. Un gestionnaire de versions est très simple à mettre en place, et tout aussi simple à utiliser. En revanche, il vous rendra des services immenses lors de la moindre évolution de votre projet.

Dans les chapitres suivants, nous verrons donc comment intégrer un gestionnaire de versions à votre tracker. Vous aurez ainsi, dès votre entrée dans l'univers de Django, un ensemble d'outils efficaces et performants qui garantiront la réussite de vos futurs projets. En somme, vous avez en main les outils indispensables à l'implémentation d'une méthode « agile » et vous allez dès maintenant expérimenter cette méthode : travail itératif, gestion du temps et gestionnaire de versions.

> **GESTION DE PROJET** **Les méthodes agiles**
>
> Les méthodes agiles sont un ensemble de techniques qui visent à être plus pragmatiques que les anciennes. Au cours du développement, elles impliquent beaucoup le demandeur (souvent le client) et se caractérisent par des cycles de livraison courts.
>
> 🕮 V. Messager, *Gestion de projet agile*, Eyrolles, 2013.

Initialisation du projet

Il est désormais temps de réutiliser ce que vous connaissez déjà pour mettre en place votre environnement de travail.

Mise en place

Dans le chapitre précédent, vous avez appris à réaliser un environnement de travail, et à y installer Django et un nouveau projet. Lancez donc les commandes que vous connaissez déjà afin de créer l'environnement adéquat pour votre tracker :

```
$ virtualenv tracker_env
(tracker_env)$ source tracker_env/bin/activate
(tracker_env)$ pip install django
(tracker_env)$ django-admin.py startproject tracker
```

Création d'une application : startapp

Maintenant, créez une nouvelle application au sein de votre projet :

```
(tracker_env)$ cd tracker
(tracker_env)$ python manage.py startapp ticket
```

Comme vous le voyez, un nouveau dossier vient d'être créé dans le répertoire de votre projet, portant le nom `ticket`.

Avant d'aller plus loin, observons comment Django a organisé l'espace de travail.

Vue d'ensemble du projet

Dans le dossier `tracker`, vous trouverez :

- `tracker/__init__.py`. Si vous avez déjà des notions de Python, vous savez que ce fichier permet à ce dossier d'être traité comme un paquet Python ;

- `manage.py`. C'est le point d'entrée vers toutes les commandes `manage` que vous utiliserez tout au long du développement de votre application ;
- `tracker/urls.py`. C'est la table de routage de votre application. Dans ce fichier, vous allez « brancher » les différentes applications de votre projet à vos URL ;
- `ticket`. Vous retrouvez bien entendu le dossier de l'application que vous venez de créer – nous verrons bientôt comment ce fichier s'organise ;
- `tracker/settings.py`. Il s'agit d'un fichier dans lequel vous retrouvez tous les paramètres nécessaires à votre application. Pour le moment, ce fichier contient déjà un ensemble de variables d'environnement qui rendent votre installation fonctionnelle, mais vous allez dès maintenant modifier ce fichier selon vos besoins spécifiques.

MÉTHODE La démarche Django

Avec Django, quelques lignes de code suffisent pour mettre en place très rapidement une application, de façon claire et découplée :

- ajouter votre application aux *settings* ;
- modifier le fichier `urls.py` à la racine de votre projet pour le faire pointer vers votre application ;
- créer un fichier `urls.py` au sein de votre application ;
- indiquer une première URL ;
- créer la vue correspondante.

Paramétrer les variables d'environnement (settings)

Les *settings* sont les variables d'environnement de votre projet, qui définissent les informations de votre base de données, les paramètres de langue, les applications que votre projet utilisera, l'emplacement de vos fichiers statiques et bien d'autres choses que nous verrons en détail.

Mode de développement (DEBUG)

/tracker/settings.py

```
DEBUG = True
TEMPLATE_DEBUG = DEBUG
```

La variable `DEBUG` lance Django en mode de développement. De nombreux outils y sont mis à votre disposition pour faciliter le développement. Par exemple, pour chaque erreur que vous rencontrerez, vous aurez accès au *backtrace* complet, qui vous aidera à trouver l'origine du problème. Il est bien entendu fortement conseillé de laisser le serveur en mode `DEBUG` pendant tout la phase de développement de votre application.

Backtrace

Lorsqu'un programme Python rencontre une erreur, il suit un cheminement au travers des différentes fonctions qui ont exécuté le code erroné. Il affiche l'ensemble de ce cheminement, ce qui vous permet de retracer à l'envers ce qui a conduit à l'erreur.

TEMPLATE_DEBUG ne peut être utile que dans le cas où la variable DEBUG est positionnée à True. En effet, cette dernière affichera une page détaillée pour chaque erreur, directement dans le navigateur (voir figure 2-1).

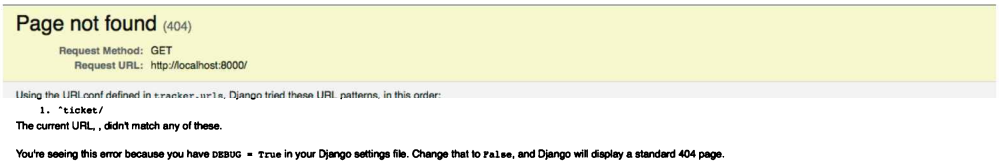

Page not found (404)
Request Method: GET
Request URL: http://localhost:8000/
Using the URLconf defined in tracker.urls, Django tried these URL patterns, in this order:
1. ^ticket/
The current URL, , didn't match any of these.
You're seeing this error because you have DEBUG = True in your Django settings file. Change that to False, and Django will display a standard 404 page.

Figure 2–1 Une page d'erreur

Configuration de la base de données

Il est temps de configurer la base de données de votre projet. Il suffit pour cela d'éditer la variable DATABASE. Voici comment se présente cette partie des variables (settings) avant modification.

Paramètre de base de données avant modification

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.',
# Add 'postgresql_psycopg2', 'postgresql', 'mysql', 'sqlite3' or 'oracle'.
        'NAME': '',
# Or path to database file if using sqlite3.
        'USER': '',
# Not used with sqlite3.
        'PASSWORD': '',
# Not used with sqlite3.
        'HOST': '',
# Set to empty string for localhost. Not used with sqlite3.
        'PORT': '',
# Set to empty string for default. Not used with sqlite3.
    }
}
```

Puisque vous serez le seul à utiliser ce tracker, au moins dans un premier temps, nous opterons pour une base de données SQLite. Django étant très tolérant en termes de base de données, je vous conseille, lors des premiers pas de votre application, d'utiliser SQLite, un SGBD performant, très simple à mettre en place et qui permet, entre autres avantages, de maintenir votre concentration sur votre code et non sur la gestion de votre base. Cette base de données sous forme de fichier se configure très simplement.

Configuration de la base de données

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
# Indique quel type de base de données utiliser.
        'NAME': 'tracker.db',
# Le fichier de base de données sera créé dans le dossier tracker.
    }
}
```

Pour communiquer avec la base de données, Django a besoin que le connecteur Python vers ce type de base soit installé sur votre machine. Il se présente sous la forme d'un module qu'il faut installer spécifiquement pour chaque type de base de données. En ce qui concerne SQLite3, le connecteur est livré en standard avec Python. Si ce dernier est installé, vous avez déjà tout ce qu'il faut pour continuer.

Internationalisation

Django propose de puissants outils d'internationalisation. Il est donc très simple de le configurer pour le faire fonctionner en français.

Tout d'abord, configurez le fuseau horaire, en utilisant les codes standards de Django dont vous trouverez une description complète, par exemple, dans la documentation en ligne de PostgreSQL.

▸ http://www.postgresql.org/docs/8.1/static/datetime-keywords.html#DATETIME-TIMEZONE-SET-TABLE

Remplacez donc :

```
TIME_ZONE = 'America/Chicago'
```

par

```
TIME_ZONE = 'Europe/Paris'
```

RESSOURCES Paramètres de langue

Pour modifier les paramètres de langue, Django s'appuie sur les codes de la norme ISO 639-1, dont vous pourrez trouver la liste ici :

- http://fr.wikipedia.org/wiki/Liste_des_codes_ISO_639-1

Dans le cas présent, vous pouvez modifier le paramètre :

```
LANGUAGE_CODE = 'en-us'
```

en

```
LANGUAGE_CODE = 'fr-fr'
```

Applications installées par défaut

Regardez le paramètre `INSTALLED_APPS` :

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
)
```

Django est livré par défaut avec quelques applications. La première, `django.contrib.auth`, va gérer toute la partie authentification de votre application. Nous aborderons plus tard les autres applications.

Il y a bien d'autres choses que vous pourrez faire avec les settings mais, pour le moment, les changements que nous venons d'appliquer sont suffisants pour bien démarrer.

Initialisation de la base de données : syncdb

Maintenant que vous avez indiqué à Django quelle base de données vous souhaitez utiliser, il faut l'initialiser avec la commande suivante :

```
[user@local]$ python manage.py syncdb
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_user_permissions
Creating table auth_user_groups
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site

You just installed Django's auth system, which means you don't have any
superusers defined.
Would you like to create one now? (yes/no):
```

Plusieurs tables sont créées par cette commande, toutes nécessaires au fonctionnement des applications qui ont été définies par Django dans la section `INSTALLED_APPS`.

À la question que Django vous pose, répondez `yes` pour qu'il crée un super-utilisateur qui vous sera utile dans un moment. Répondez ensuite à toutes les questions (nom d'utilisateur, adresse électronique, mot de passe) :

```
Would you like to create one now? (yes/no): yes
Username (Leave blank to use 'user'):
E-mail address: user@exemple.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
No fixtures found.
```

Si vous observez le contenu de votre répertoire `tracker`, vous vous apercevrez que Django a créé pour vous le fichier `tracker.db` qui accueille votre base de données.

Projet versus application

Comme nous l'expliquions plus haut, Django découpe votre projet en plusieurs applications. Avec la commande `python manage.py startapp ticket`, vous avez déjà créé la première. Observons maintenant ce que Django a créé pour vous :

```
[user@local]$ ls ticket
__init__.py models.py tests.py views.py
```

Voyons à quoi ces éléments correspondent.

- `__init__.py` : comme son homologue présent dans le dossier `tracker`, ce fichier permet à Python de considérer ce dossier comme un paquet.
- `models.py` : ce fichier fait le lien entre votre application et la base de données. Il correspond tout simplement au « M » du sigle MVC (*Model-View-Controler*, ou modèle-vue-contrôleur). Nous en reparlerons en détail au chapitre 10 « Django et les bases de données ».
- `test.py` : c'est un fichier que vous utiliserez pour lancer vos batteries de tests, dans le cas très conseillé où vous mettiez en pratique le principe de TDD (*Test Driven Development*, ou développement dirigé par les tests).
- `views.py` : il s'agit du cœur de votre application. Les vues *(views)* vont accueillir l'essentiel de votre code. Nous en reparlerons longuement au chapitre 11 « Le traitement de l'information : les vues ».

Configuration des applications installées

Il faut indiquer à Django que vous souhaitez utiliser cette application pour votre projet `tracker`. Ouvrez tout simplement le fichier `settings.py` et ajoutez votre application à la liste des `INSTALLED_APPS` :

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'ticket',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
)
```

Les URL

Il faut ensuite indiquer à Django, dans le fichier `urls.py`, l'URL racine qui pointera vers votre application.

Fichier urls.py

```
from django.conf.urls.defaults import patterns, include, url

# Uncomment the next two lines to enable the admin:
# from django.contrib import admin
# admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'tracker.views.home', name='home'),
    # url(r'^tracker/', include('tracker.foo.urls')),

    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # url(r'^admin/', include(admin.site.urls)),
)
```

Décommentez la ligne :

```
# url(r'^tracker/', include('tracker.foo.urls')),
```

et modifiez-la comme suit :

```
    url(r'^ticket/', include('tracker.ticket.urls')),
```

Elle va simplement indiquer à Django que toutes les URL commençant par `/ticket/` doivent pointer vers les URL définies dans le module `urls`, fichier `urls.py` du paquet `ticket`. Cette façon de procéder est très pratique car elle permet de bénéficier de l'héritage dans les URL. Elle vous aide à découper ces dernières de façon claire et modulaire, puisque chaque application viendra compléter dans son propre fichier `urls.py` le point d'entrée défini dans le fichier `urls.py` à la racine du projet.

> **Méthode Développement dirigé par les erreurs**
>
> Les erreurs font partie intégrante du développement d'une application web. L'une des clés vers la maîtrise du framework Django est de savoir reconnaître celles que vous allez rencontrer et comment les corriger. C'est pourquoi, dans les exemples qui suivent, vous allez être confronté à quelques cas d'erreurs et apprendrez à les corriger.

Pour le moment, le fichier `ticket/urls.py` n'existe pas. Il faut donc le créer.

Création du fichier ticket/urls.py

```
from django.conf.urls.defaults import patterns, url
from views import home

urlpatterns = patterns('',
    url(r'^home/$', home, name="home")
)
```

Tout d'abord, importez les fonctions dont vous avez besoin depuis Django (ici, `patterns` et `url`). Ensuite, importez la fonction `home` du fichier `views` (cette fonction n'existe pas encore, mais vous allez la créer dans un instant). Enfin, créez un nouveau `urlpatterns` qui ne va contenir qu'une URL :

```
url(r'^home/$',home, name = "home")
```

Cette ligne indique à Django que l'URL correspondant à `/ticket/home/` doit pointer vers la fonction `home` que vous venez d'importer du ficher `views`.

Testez ces quelques modifications. Lancez votre serveur de développement :

```
python manage.py runsever
```

et rendez-vous sur la page `http://localhost:8000` (voir figure 2-2).

Figure 2–2 Message d'erreur

Une première vue

Django signale l'absence de la fonction `home`. Vous devez donc la créer dans le fichier `ticket/views.py`.

Fichier ticket/views.py

```
def home(request):
    print "hello world"
```

Voici une fonction du fichier `views.py` dans sa forme la plus simple (dans le reste du livre, nous utiliserons le terme « vue » pour parler des fonctions du fichier `views.py`). Elle prend en paramètre l'objet `request` créé par Django, dans lequel vous trouverez de nombreuses informations – nous y reviendrons plus loin.

Testez de nouveau votre projet en vous rendant sur la page `http://localhost:8000`.

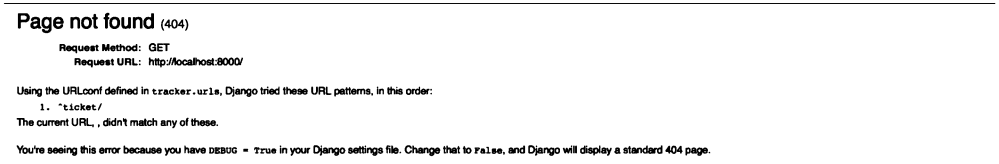

Page not found (404)
Request Method: GET
Request URL: http://localhost:8000/
Using the URLconf defined in tracker.urls, Django tried these URL patterns, in this order:
1. ^ticket/
The current URL, , didn't match any of these.
You're seeing this error because you have DEBUG = True in your Django settings file. Change that to False, and Django will display a standard 404 page.

Figure 2–3 La page n'est pas trouvée.

Comme vous le remarquez, vous n'obtenez plus l'écran de la figure 2-2, mais un message qui vous indique que la page recherchée n'existe pas. Un coup d'œil dans votre terminal vous confirmera d'ailleurs cette erreur.

```
[28/Jul/2011 23:32:03] "GET / HTTP/1.1" 404 1983
```

Le code `404` est renvoyé lorsque le serveur ne trouve pas la page demandée.

Les codes HTTP

Le protocole HTTP, entre autres choses, définit un ensemble de *status codes*. Il s'agit de codes renseignant sur le résultat global d'une requête. Les plus courants sont 200 (succès), 404 (ressource non trouvée) et 500 (erreur interne du serveur).

Fort heureusement, sur la page d'erreur, Django indique qu'il a dans ses URL un pattern commençant par `^ticket`. Essayez de charger la page `http://localhost:8000/ticket/` (voir figure 2-4).

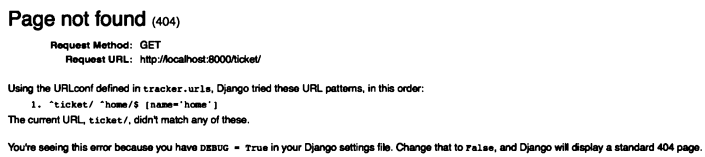

Page not found (404)
Request Method: GET
Request URL: http://localhost:8000/ticket/
Using the URLconf defined in tracker.urls, Django tried these URL patterns, in this order:
1. ^ticket/ ^home/$ [name='home']
The current URL, ticket/, didn't match any of these.
You're seeing this error because you have DEBUG = True in your Django settings file. Change that to False, and Django will display a standard 404 page.

Figure 2–4 Ce n'est toujours pas la bonne URL.

Encore une fois, Django vous donne de précieuses informations. S'il ne trouve toujours pas la page que vous lui demandez, il vous indique en revanche qu'il existe un chemin `http://localhost:8000/ticket/home/`. Rendez-vous donc à cette adresse (voir figure 2-5).

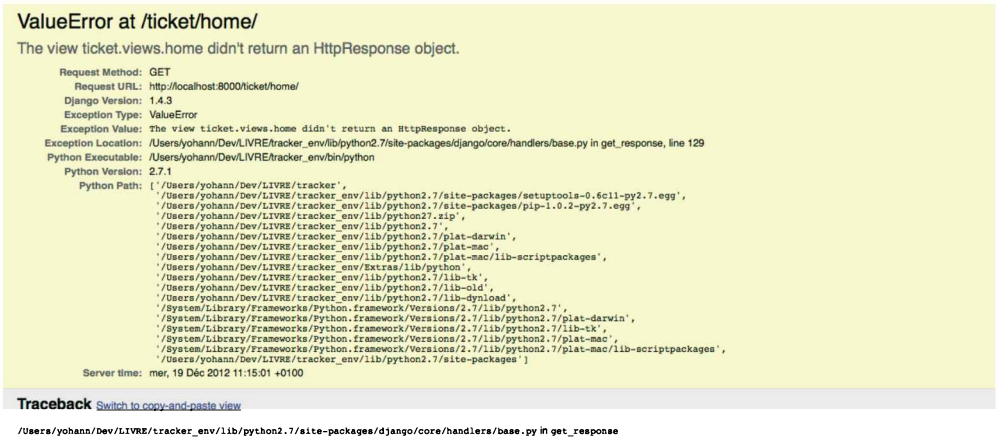


Figure 2–5 Il manque un objet de type HttpResponse.

Encore une erreur ! Cette fois, Django précise que la vue `tracker.ticket.views.home` n'a pas renvoyé d'objet de type `HttpResponse`. Nous verrons plus loin comment créer ce type d'objet mais, pour le moment, tournez-vous vers votre terminal :

```
Hello World!
[29/Jul/2011 00:05:15] "GET /ticket/home/ HTTP/1.1" 500 55420
```

Vous constatez que votre fonction a eu, en réalité, le résultat escompté : bien que Django ne puisse pas afficher une page dans le navigateur en raison de l'objet `HttpResponse` manquant, il a tout de même affiché le message `Hello World!` dans le terminal. Cette fonctionnalité vous permettra tout au long du développement de votre application de tester, de déboguer et de profiter des capacités d'introspection qu'offrent Python et Django.

L'objet request

Revenons à l'objet `request` que vous avez découvert dans la fonction `home`. Pour en savoir un peu plus sur lui, modifiez cette dernière.

```
# Create your views here
def home(request):
    print request
```

Rechargez la page `http://localhost:8000/ticket/home/`. La page renvoyée par Django affiche encore le même message d'erreur, ce qui est parfaitement normal puisque vous ne lui fournissez toujours pas d'objet `HttpResponse`. Cependant, le terminal vous indique de nouveaux éléments :

```
<WSGIRequest
GET:<QueryDict: {}>,
POST:<QueryDict: {}>,
COOKIES:{},
...<truncated>
'HTTP_ACCEPT': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/
*;q=0.8',
'HTTP_ACCEPT_CHARSET': 'ISO-8859-1,utf-8;q=0.7,*;q=0.7',
'HTTP_ACCEPT_ENCODING': 'gzip, deflate',
'HTTP_ACCEPT_LANGUAGE': 'fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3',
'HTTP_CONNECTION': 'keep-alive',
'HTTP_HOST': 'localhost:8000',
'HTTP_USER_AGENT': 'Mozilla/5.0 (X11; Linux i686; rv:5.0) Gecko/20100101
Firefox/5.0',
...<truncated>
```

Vous avez dans l'objet `request` toute la requête qui a été transmise au serveur : les paramètres `GET` et `POST`, les en-têtes HTTP, etc. C'est grâce à cet objet que vous serez en mesure de dialoguer avec l'utilisateur tout au long de la vie de l'application.

L'objet HttpResponse

Comme toute requête appelle une réponse, voyons maintenant comment renvoyer au navigateur l'objet `HttpResponse` réclamé par Django. Ouvrez votre fichier `views.py` et modifiez-le comme suit :

```
from django.http import HttpResponse
def home(request):
    return HttpResponse("Hello world!")
```

La seule différence par rapport à l'exemple précédent est d'avoir importé la classe `HttpResponse` et de l'avoir instanciée avec un message.

Rendez-vous encore une fois à l'adresse `http://localhost:8000/ticket/home/` (voir figure 2-6). Comme vous le constatez, votre page affiche désormais votre message.

Figure 2–6
Une première page fonctionnelle

RESSOURCE Sur le gestionnaire de versions

Pour vous simplifier la vie et vous aider à détecter vos erreurs, dans le cas où vous en rencontreriez encore à cette étape de la vie de votre application, vous pouvez récupérer une copie fonctionnelle de ce code sur Bitbucket (https://www.bitbucket.org) en utilisant la commande `hg clone https://bitbucket.org/boblefrag/livre-django-tracker`. Nous indiquerons à chaque étape la révision que vous devrez utiliser pour que les exemples de code correspondent au livre.

RAPPEL La démarche Django

Récapitulons la démarche à adopter avec Django pour mettre en place très rapidement une application, de façon claire et découplée : ajouter l'application aux settings, modifier le fichier `urls.py` à la racine du projet pour le faire pointer vers votre application, créer un fichier `urls.py` au sein de l'application, indiquer une première URL et créer la vue correspondante.

Créer une application dynamique

L'exemple que vous venez de créer est pleinement fonctionnel, mais il lui manque ce qui fait qu'une application web est plus qu'un simple site statique. Heureusement, tout est en place pour qu'avec quelques modifications, votre application statique devienne dynamique.

Imaginons cette fois qu'au lieu d'`Hello World!` nous souhaitions afficher le nom d'une personne, par exemple `Hello Pierre!`.

Commencez par modifier le fichier `urls.py` de votre application `ticket` :

```
from django.conf.urls.defaults import patterns, url
from views import home
urlpatterns = patterns('',
    url(r'^home/(\w+)$', home, name="home")
                       )
```

Vous venez d'ajouter `(\w+)`, qui est une expression régulière indiquant à Django de capturer toutes les lettres après le / et de transmettre à la vue la chaîne de caractères ainsi constituée.

Pour l'heure, intéressez-vous au pattern que vous venez de définir : `w+`.

- `w` indique un caractère alphabétique. C'est une écriture raccourcie de `[Aa-Zz]`.
- `+` signifie aucune, une fois ou plusieurs fois le type de caractère précédent.

`w+` correspondra donc à `a`, `aa`, `ba`, mais pas à `1`, `a1`, `1e`, `aerrty-jhz`, etc.

Dans notre exemple, les URL qui seront routées vers notre vue pourront être :

- `http://localhost:8000/ticket/home/pierre`
- `http://localhost:8000/ticket/home/Untest`

mais pas :

- `http://localhost:8000/ticket/home/Un-test`
- `http://localhost:8000/ticket/home/un_test`

Testez maintenant votre application en saisissant l'une des deux premières URL. Vous devriez rencontrer l'erreur ci-contre (voir figure 2-7).

Django a bien fait passer l'argument que vous avez défini dans votre URL vers la vue `home`. Seulement, celle-ci prend `request` comme seul argument. Or, avec la modification que vous venez d'effectuer dans vos URL, vous lui avez fourni un nouvel argument dont elle ne sait que faire.

TypeError at /ticket/home/test

home() takes exactly 1 argument (2 given)

Request Method:	GET
Request URL:	http://localhost:8000/ticket/home/test
Django Version:	1.4.3
Exception Type:	TypeError
Exception Value:	home() takes exactly 1 argument (2 given)
Exception Location:	/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/site-packages/django/core/handlers/base.py in get_response, line 111
Python Executable:	/Users/yohann/Dev/LIVRE/tracker_env/bin/python
Python Version:	2.7.1
Python Path:	['/Users/yohann/Dev/LIVRE/tracker', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/site-packages/setuptools-0.6c11-py2.7.egg', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/site-packages/pip-1.0.2-py2.7.egg', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python27.zip', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/plat-darwin', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/plat-mac', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/plat-mac/lib-scriptpackages', '/Users/yohann/Dev/LIVRE/tracker_env/Extras/lib/python', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/lib-tk', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/lib-old', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/lib-dynload', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-tk', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/lib-scriptpackages', '/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/site-packages']
Server time:	mer, 19 Déc 2012 11:39:41 +0100

Figure 2–7 Le nombre d'arguments n'est pas correct.

MÉTHODE Tester les URL via la vue

Vous vous demandez peut-être pourquoi nous souhaitons tester une vue qui va échouer. La raison est fort simple et nous vous invitons à suivre le même cheminement de développement : en vérifiant que la vue échoue, vous vous assurez que l'URL a bien fourni un argument supplémentaire à votre vue. C'est donc les URL que vous testez en levant cette erreur, pas la vue.

Django étant un framework itératif, utilisez-le au maximum et essayez d'atteindre la granularité la plus fine possible. Cela vous garantit de valider et de tester chaque modification avant de développer la suivante. Vous avancez ainsi sur un terrain balisé et, lorsqu'un problème se pose, vous n'avez pas à chercher pendant des heures d'où provient votre erreur.

Modifiez donc votre vue pour qu'elle prenne ce second argument :

```
from django.http import HttpResponse
def home(request, name):
    return HttpResponse("Hello %s " % name)
```

Puis rechargez votre navigateur, pour obtenir l'écran représenté sur la figure 2-8.

Figure 2–8
Un Hello World! dynamique

Et voilà, votre application est désormais pleinement dynamique. Seulement, l'adresse `http://localhost:8000/home/` ne fonctionne plus.

La première chose qui pourrait vous venir à l'idée serait de créer deux URL différentes et deux vues pour chacune des URL :

```
from django.conf.urls.defaults import patterns, url
from views import home, home_name

urlpatterns = patterns('',
    url(r'^home/$', home, name="home"),
    url(r'^home/(\w+)/$', home_name, name="home_name")
                    )
                    )
```

et dans le fichier `views.py` :

```
from django.http import HttpResponse


def home(request):
    return HttpResponse("Hello World!")


def home_name(request, name):
    return HttpResponse("Hello %s " % name)
```

Voyons si nous ne pouvons pas factoriser un peu plus :

```
from django.conf.urls.defaults import patterns, url
from views import home

urlpatterns = patterns('',
    url(r'^home/$', home, name="home"),
    url(r'^home/(\w+)$', home, name="home")
                    )
```

Vous avez ainsi deux URL pointant vers la même fonction `home`.

Spécifiez dans votre vue un paramètre optionnel :

```
from django.http import HttpResponse

def home(request, name=None):
# Vérifions l'existence de l'attribut name.
    if name:
        return HttpResponse("Hello %s " % name)
    else:
        return HttpResponse("Hello World!")
```

De cette façon, suivant l'URL que vous indiquerez, le serveur vous renverra l'une ou l'autre des réponses. Vous verrez que cette méthode très simple est extrêmement puissante, et qu'elle vous sera très utile lorsque vous aurez besoin de traiter des requêtes Ajax ou de renvoyer du PDF par exemple.

Gestionnaire de versions : révision 1

Pour retrouver le code comme il devrait être à cette étape, récupérez le code présent sur Bitbucket et mettez-le à jour à la révision 1 avec la commande `hg up -r 1`.

Le modèle de l'objet ticket

Vous venez de mettre en place une première vue dynamique. Si vous connaissez déjà bien le principe du MVC, vous constaterez que nous n'avons fait qu'effleurer le « C » (contrôleur) avec les vues et que nous avons introduit le principe de routage avec les URL. Maintenant, il est temps de rentrer dans le vif du sujet en définissant notre premier modèle.

Pour votre application `ticket`, vous allez avoir besoin d'un objet tâche qui représentera les choses à faire. Vous pouvez le définir comme ceci :

- un nom pour la retrouver facilement ;
- une description de la tâche à accomplir ;
- la date de création ;
- la date à laquelle cette tâche doit être terminée *(due-date)* ;
- la date à laquelle vous aimeriez commencer à travailler sur cette tâche *(schedule-date)* ;
- l'état du ticket : ouvert ou fermé – un ticket fermé étant une tâche réalisée.

Commencez donc à réfléchir aux types dont vous allez avoir besoin :

- un nom : une chaîne de caractères ;
- une description : un texte ;
- la date de création du ticket : une date ;
- la due-date : une date ;
- la schedule-date : une date ;
- fermé ou ouvert : un booléen.

La classe Task

Il suffit d'ouvrir le fichier `models.py` et de créer une classe `Task` comme suit :

Création de la classe Task

```
from django.db import models

class Task(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()
    created_date = models.DateField(auto_now_add=True)
    due_date = models.DateField()
    schedule_date = models.DateField()
    closed = models.BooleanField(default=False)
```

La classe que vous venez de créer hérite de la classe `Model`, et chaque attribut de votre classe est affecté à une classe du module `models` qui définit son type. Dans le cas présent, vous utilisez les champs de types suivants.

- `CharField` sera interprété au niveau de la base de données comme une colonne de type `string` (c'est-à-dire une chaîne de caractères). Nous lui ajoutons le paramètre `max_length` qui adjoindra la contrainte `MAX_LENGTH` (longueur maximale) à 250 caractères dans la base de données.
- `TextField` (un texte sur plusieurs lignes) sera interprété au niveau de la base de données comme une colonne de type `text`.
- `DateField` (un objet de type date) sera traduit, au niveau de la base de données, comme un objet de type `date` et, au niveau de Django, comme un objet de type `datetime.datetime`. Le paramètre optionnel `auto_now_add` permet l'autocomplétion de l'attribut à la date du jour lors de sa création.
- `BooleanField` (un objet de type booléen) sera traduit, au niveau de la base de données, comme un objet de type `Boolean`. Le paramètre `default` place l'état fermé de la tâche à « Faux » ; la tâche est donc ouverte.

Pour le moment, ces quelques informations sont suffisantes pour commencer à utiliser votre application. Enregistrez votre fichier `models.py` et indiquez à Django qu'il lui faut créer la table `task`.

Création de la table task

```
[user@local]$ python manage.py syncdb
Creating tables ...
Creating table ticket_task
Installing custom SQL ...
Installing indexes ...
No fixtures found.
```

Avec cette simple commande, Django vient de créer pour vous la table `task` (tâche) et toutes les colonnes que vous avez définies dans le fichier `models.py`.

La commande python manage.py shell

Pour le vérifier, utilisez la commande `python manage.py shell`, très pratique.

```
[user@local]$ python manage.py shell
Python 2.7.1 (r271:86832, Apr 12 2011, 16:16:18)
[GCC 4.6.0 20110331 (Red Hat 4.6.0-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from ticket.models import Task# Importez le modèle Task dans la console
interactive.
>>> Task.objects.all()# Listez tous les objets Task présents dans la base de
données.
[]
```

La réponse que vous renvoie Django est une liste vide, car vous n'avez ouvert aucun ticket pour le moment. Vous allez maintenant essayer d'enregistrer une nouvelle tâche. Comme vous le remarquerez, Django veille sur vous et ne vous permet pas de créer des objets si certaines données sont manquantes.

Création d'une tâche

```
>>> tache = Task()# On instancie un objet vide de la classe Task.
>>> tache.name = "Ma première tâche" # On lui donne un nom
>>> tache.description = "La première tâche d'une longue série" # une description
>>> tache.save() # Essayons de l'enregistrer.
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/usr/lib/python2.7/site-packages/django/db/models/base.py", line 460,
in save
    self.save_base(using=using, force_insert=force_insert,
force_update=force_update)
  File "/usr/lib/python2.7/site-packages/django/db/models/base.py", line 553,
in save_base
    result = manager._insert(values, return_id=update_pk, using=using)
  File "/usr/lib/python2.7/site-packages/django/db/models/manager.py", line
195, in _insert
    return insert_query(self.model, values, **kwargs)
  File "/usr/lib/python2.7/site-packages/django/db/models/query.py", line 1436,
in insert_query
    return query.get_compiler(using=using).execute_sql(return_id)
  File "/usr/lib/python2.7/site-packages/django/db/models/sql/compiler.py",
line 791, in execute_sql
    cursor = super(SQLInsertCompiler, self).execute_sql(None)
  File "/usr/lib/python2.7/site-packages/django/db/models/sql/compiler.py",
line 735, in execute_sql
```

```
    cursor.execute(sql, params)
  File "/usr/lib/python2.7/site-packages/django/db/backends/util.py", line 34,
in execute
    return self.cursor.execute(sql, params)
  File "/usr/lib/python2.7/site-packages/django/db/backends/sqlite3/base.py",
line 234, in execute
    return Database.Cursor.execute(self, query, params)
IntegrityError: ticket_task.due_date may not be NULL
```

Encore une fois, la réponse de Django est on ne peut plus claire :

```
IntegrityError: ticket_task.due_date may not be NULL
```

Il refuse d'enregistrer le ticket que vous venez de créer, car vous n'avez pas précisé de `due_date`. C'est une fonctionnalité très puissante de Django, puisqu'il s'occupe à votre place de toute la partie validation de données ; elle vous sera particulièrement utile, par exemple, lorsque vous devrez créer des formulaires.

Renseignez tous les champs nécessaires à l'enregistrement de votre objet `tache`.

Enregistrement de l'objet task

```
>>> import datetime
# Comme la due-date est de type date, nous importons le module standard datetime
de python.
>>> tache.due_date = datetime.datetime.now() # Une due_date à aujourd'hui
>>> tache.schedule_date = datetime.datetime.now() # Une schedule_date à
aujourd'hui également.
>>> tache.save() # Tous les champs sont maintenant remplis, vous pouvez
enregistrer votre ticket.
>>> Task.objects.all() # Listez tous les objets Task.
[<Task: Task object>]
>>> tache # Vérifiez que les champs autoremplis par Django sont bien exacts.
<Task: Task object> # Hum, ce n'est pas très instructif !
>>> tache.name
'Ma première tâche'
>>> tache.created_date
datetime.date(2011, 7, 29)
>>> tache.closed
False
```

Vous venez de créer le modèle `Task` et de voir comment le manipuler au travers de la console interactive fournie par Django. Vous utiliserez régulièrement cette console dès que vous aurez besoin d'en savoir plus sur un objet, ou tout simplement pour réaliser quelques tests avant d'inclure un modèle dans le flux de votre application. Nous allons maintenant vous montrer à quel point Django peut vous simplifier la vie avec l'une de ses applications les plus puissantes : l'interface d'administration.

Première itération : l'interface d'administration

L'interface d'administration de Django est une application au même titre que l'application `ticket` que vous êtes en train de réaliser. Elle vous permet de créer, de modifier, de supprimer et de lister tous les objets de vos modèles, dans le navigateur.

> Nous montrerons plus loin dans ce livre comment la configurer très finement.

Pour activer l'interface d'administration, vous devez reprendre le cheminement que vous connaissez maintenant bien car vous l'avez déjà expérimenté avec l'application `ticket` :

1. Ajouter l'application aux settings.
2. Créer un lien dans les URL à la racine de votre projet vers votre application.

Tout le reste se fait de façon automatique, bien entendu.

Ajouter l'application aux settings

En réalité, Django vous simplifie le travail au maximum (c'est le principe d'un framework) ; il a déjà balisé le chemin pour vous. Il vous suffit donc, pour activer l'interface d'administration dans les settings, de décommenter la ligne :

```
# 'django.contrib.admin'
```

La partie `INSTALLED_APPS` des settings doit donc ressembler à cela :

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'ticket',
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
)
```

Ajouter l'application aux URL

Si vous avez bien suivi le processus de mise en place d'une nouvelle application Django dans votre projet, vous connaissez déjà la prochaine étape : elle consiste à ajouter un chemin d'URL depuis votre projet vers le fichier URL de votre nouvelle application. Là encore, Django a balisé le chemin pour vous. Décommentez les lignes suivantes.

Fichier tracker/urls.py

```
# from django.contrib import admin
# admin.autodiscover()
# url(r'^admin/', include(admin.site.urls))
```

Un premier test

Vous avez maintenant tout ce qu'il vous faut pour tester l'interface d'administration. Lancez votre serveur de test :

```
python manage.py runserver
```

Puis rendez-vous à l'adresse `http://localhost:8000/admin/` (voir figure 2-9). Le nom d'utilisateur et le mot de passe sont ceux que vous avez définis lorsque vous avez lancé la commande `python manage.py syncdb`.

Figure 2–9
Authentification lors de la connexion à l'interface d'administration

Une fois connecté, voici l'écran que vous allez découvrir (voir figure 2-10).

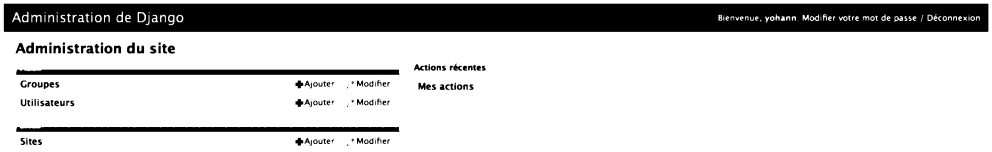


Figure 2–10 Écran d'accueil de l'interface d'administration

Visiblement, si l'interface d'administration est bien fonctionnelle pour les applications `auth` et `site`, il n'y a aucune trace de l'application que vous venez de créer. Là encore, ce n'est pas une erreur mais une fonctionnalité très utile de Django. En effet, il vous laisse la possibilité de choisir très simplement les applications que vous souhaitez rendre disponibles dans l'interface d'administration. Nous verrons bientôt que cette finesse de configuration va bien au-delà du niveau « application » puisqu'il est également possible de sélectionner les modèles que vous souhaitez rendre disponibles sur l'interface d'administration.

Ajouter le modèle Task à l'administration

Pour ajouter le modèle `Task` à l'interface d'administration, il suffit de le signifier à Django via le fichier `admin.py` de chaque application. Voici une version très simple, mais fonctionnelle, de ce fichier.

Fichier admin.py

```
# Importez tout d'abord l'application admin dans ce fichier.
from django.contrib import admin
# Importez le modèle que vous souhaitez rendre disponible.
from models import Task

# Indiquez à Django que vous souhaitez le rendre disponible dans l'interface
d'administration.
admin.site.register(Task)
```

Rechargez votre navigateur pour y voir apparaître les changements (voir figure 2-11).

Figure 2–11 L'application ticket apparaît désormais.

L'application `ticket` apparaît désormais sur l'écran d'accueil de Django et le modèle `Task` peut être géré entièrement depuis l'interface d'administration.

Gestionnaire de versions : révision 2

Retrouvez cette étape à la révision 2 du projet.

Une interface CRUD

L'interface d'administration est entièrement CRUD (*Create, Read, Update, Delete*, soit « créer, lire, modifier, supprimer »).

Un premier aperçu

Rendez-vous à l'adresse `http://localhost:8000/admin/ticket/` (voir figure 2-12). Sur cette page sont listés tous les modèles de l'application `ticket`. Pour le moment, seul un modèle est disponible, le modèle `Task`.

Figure 2–12 Administration de l'application ticket

À l'adresse `http://localhost:8000/admin/ticket/task` sont listés tous les objets du modèle `Task`. Pour le moment, seul un objet est présent : celui que vous avez créé via `python manage.py shell`. En cliquant sur son nom, *Task object*, vous accédez à un formulaire qui vous permet de le modifier (voir figure 2-13).

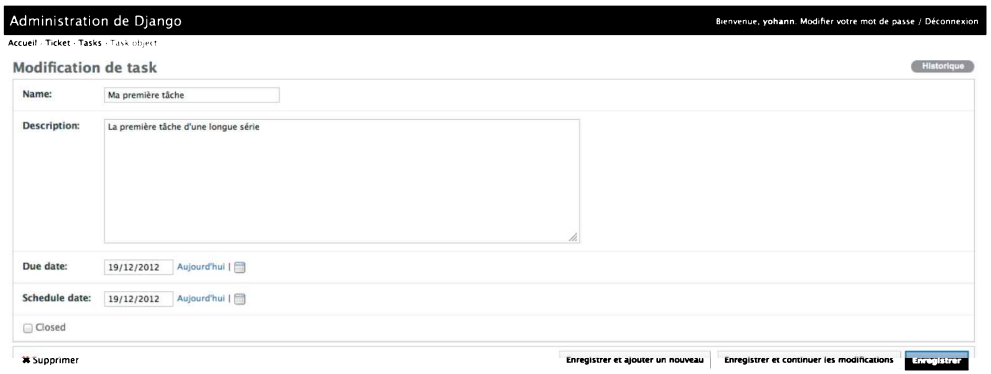


Figure 2–13 Le formulaire d'édition

Le lien *Supprimer*, qui pointe vers `http://localhost:8000/admin/ticket/task/1/delete/`, efface le ticket que vous aviez créé. Enfin, le lien `http://localhost:8000/admin/ticket/task/add/` propose, au contraire, un formulaire d'édition pour créer de nouveaux tickets.

Vous avez donc une interface 100 % CRUD :

- `http://localhost:8000/admin/ticket/task/add/` : Create ;
- `http://localhost:8000/admin/ticket/ticket/` : Read ;
- `http://localhost:8000/admin/ticket/ticket/1/` : Update ;
- `http://localhost:8000/admin/ticket/ticket/1/delete/` : Delete.

Comment sont créées les URL de l'administration ?

Pour vous aider à vous y retrouver facilement, voici comment sont définies en interne les URL de l'administration de Django.

- `http://localhost:8000/admin/<app>/` (où `app` correspond à l'application) donne accès à la liste des modèles de l'application `ticket`.
- `http://localhost:8000/admin/<app>/<model>/` (où `model` correspond au modèle auquel vous souhaitez accéder) liste tous les objets du modèle choisi.
- `http://localhost:8000/admin/<app>/<model>/add/` fournit un formulaire d'édition pour créer un nouvel objet de type `app`.
- `http://localhost:8000/admin/<app>/<model>/<id>/` (où `id` correspond à l'identifiant numérique unique que Django attribue automatiquement à chaque objet nouvellement créé) fournit un formulaire d'édition de l'objet référencé par `id`.
- `http://localhost:8000/admin/<app>/<model>/<id>/delete/` supprime l'objet ayant l'id `id` du modèle `model` de l'application `app`.

Les formulaires de l'administration

Comme vous le remarquez, les formulaires de l'administration de Django présentent quelques particularités. Par exemple, le champ `schedule_date` propose un calendrier JavaScript très pratique pour indiquer une date, et le champ `description` est de type texte permettant d'entrer une description longue.

Django utilise les capacités d'introspection de Python et va lire le type des champs que vous avez définis dans le fichier `models.py` afin de vous fournir le type de champ de formulaire qui semble le plus pratique. Bien entendu, il vous est tout à fait possible de définir vous-même le type de champ que vous souhaitez utiliser et même de créer vos propres types. Nous en parlerons plus longuement au chapitre 15 consacré aux contribs Django.

Modifier l'interface d'administration

Dès à présent, vous pouvez très simplement rendre l'interface d'administration de Django plus conviviale. Vous avez sans doute remarqué que sur la page `http://localhost:8000/admin/ticket/task/` tous les objets listés portent le nom `Task object`.

Cela est dû au fait que vous n'avez pas indiqué à Django comment nommer dans l'interface d'administration les objets de type `ticket`. Corrigeons cela tout de suite.

Il serait intéressant d'utiliser le champ `name` et d'avoir ainsi une liste classée par noms. Comme cette fonctionnalité pourra vous être utile dans de nombreux points de votre application, et comme vous souhaitez sans doute garder une logique DRY (*Don't Repeat Yourself*, c'est-à-dire ne préciser une information qu'à un seul point de votre application), nous vous proposons d'indiquer le nom de vos objets dans le fichier `models.py` en définissant une nouvelle méthode dans votre classe `Ticket`.

La fonction __unicode__

La méthode de `model` `__unicode__` sera lue par l'application `admin` de Django pour vous donner une liste des objets `Task` classés par ce nom. Voici comment la définir :

```
def __unicode__(self) :
    return self.name
```

On peut difficilement imaginer plus simple ! Voici à quoi ressemble désormais votre classe `Ticket` :

```
from django.db import models
class Ticket(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()
    created_date = models.DateField(auto_now_add = True)
    due_date = models.DateField()
    schedule_date = models.DateField()
    closed = models.BooleanField()

    def __unicode__(self):
        return self.name
```

Si vous vous rendez à nouveau sur la page `http://localhost:8000/admin/ticket/task/`, vous constaterez que vous avez désormais une liste beaucoup plus conviviale (voir figure 2-14).

Figure 2–14 L'interface d'administration après la modification

Gestionnaire de versions : révision 3

Retrouvez cette étape à la révision 3 du projet.

Aller plus loin dans la configuration de l'administration

Vous avez maintenant une interface d'administration utilisable. Toutefois, elle manque encore de quelques fonctionnalités qui la rendraient vraiment utile pour votre tracker. Par exemple, vous aimeriez sans doute voir les due-dates et les schedule-dates dans la liste de vos tickets. Vous pourriez même imaginer que les tickets approchant la due-date soit colorés en orange et que ceux dont la due-date est dépassée apparaissent en rouge. Vous pourriez vouloir indiquer le nombre de jours restant avant la due-date. C'est ce que nous allons voir maintenant.

Afficher les due-dates sur l'interface d'administration

Cette fois, vous n'allez pas modifier le fichier `models.py` mais le fichier `admin.py`. En effet, c'est la logique d'affichage dans l'administration que vous souhaitez changer. Il est parfois un peu compliqué de savoir si les modifications doivent être effectuées dans l'`admin` ou dans les `models` mais, avec un peu d'habitude, vous intégrerez rapidement ces réflexes.

Lorsque, comme ici, vous souhaitez modifier le comportement d'un modèle dans l'interface d'administration, il vous faut créer une classe héritant de `admin`, `ModelAdmin` :

```
class TaskAdmin(admin.ModelAdmin):
```

Plusieurs attributs sont alors disponibles. Pour l'heure, le premier dont vous allez avoir besoin est `list_display`. C'est un tuple qui précise les champs que vous souhaitez voir apparaître dans la liste des objets de votre modèle. Vous pouvez donc écrire :

```
class TaskAdmin(admin.ModelAdmin):
    list_display = ('name', 'schedule_date', 'due_date')
```

Puis, pour indiquer à Django qu'il doit utiliser cette classe pour représenter le modèle `Task` via votre classe `TaskAdmin`, vous devez modifier la ligne :

```
admin.site.register(Task)
```

en

```
admin.site.register(Task, TaskAdmin)
```

Rechargez ensuite votre navigateur et vous obtiendrez le listing suivant (voir figure 2-15).

Figure 2–15 Le listing avec les due-dates et les schedule-dates

Changer la couleur des champs due_date et schedule_date

Pour changer la couleur des champs `due_date` et `schedule_date`, voici une petite astuce.

Avec Django, vous avez la possibilité de créer des méthodes dans vos modèles qui vont se comporter comme des attributs du modèle au même titre que les champs `name` ou `description`. Vous allez donc créer une fonction `colored_due_date` qui va colorer la due-date en fonction du temps qui reste avant sa réalisation. Choisissez, par exemple, la couleur verte pour une tâche à réaliser dans plus d'une semaine, orange pour celle à réaliser dans la semaine en cours et rouge pour les due-date dépassées.

Essayez maintenant de coder cette fonction dans la classe `Ticket` et de ne lire ce qui va suivre que pour corriger votre travail. Pour vous aider, sachez que si vous souhaitez ajouter du HTML non échappé dans l'administration, vous devez marquer votre fonction avec l'attribut `allow_tags` :

```
def ma_fonction(self) :
    return "<ahref= 'http://www.exemple.com'>un lien</a>"
ma_fonction.allow_tags = True
```

Figure 2–16 Le résultat final

Voici comment vous auriez pu coder cette fonction :

```
    def colored_due_date(self):
        if self.due_date - timedelta(days=7) > date.today():
            color = "green"
        elif self.due_date + timedelta(days=7) > date.today():
            color = "orange"
        else:
            color = "red"
        return " <span style=color:%s>%s</span>" % (color, self.due_date)
    colored_due_date.allow_tags = True
```

Il vous faut, bien entendu, ajouter ce nouveau champ à l'attribut `list_display` de la classe `TicketAdmin` dans le fichier `admin.py` et peut-être supprimer le champ `due_date` qui fait maintenant doublon.

Vous remarquerez alors que, même si votre due_date est désormais colorée, elle ne s'affiche plus de manière aussi conviviale qu'avant. Cela vient du fait que Django formate automatiquement les objets dates dans votre langue, pour votre confort. Or, maintenant, c'est un objet « chaîne de caractères » que nous lui fournissons. À vous donc de formater correctement la date avant l'affichage. La méthode `strftime` est alors particulièrement indiquée :

```
due_date = self.due_date.strftime("%d %B %Y")
```

Seulement, avec ce code, vous perdez les puissantes méthodes d'internationalisation que vous offre Django. L'astuce ici est d'importer une fonction qui est à l'origine destinée aux templates pour afficher la date correctement :

```
from django.template.defaultfilters import date as django_date
due_date = django_date(self.due_date, "d F Y")
```

Votre fichier `models.py` doit maintenant ressembler à ceci :

```
from django.db import models
from datetime import date, timedelta
from django.template.defaultfilters import date as django_date


class Task(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()
    created_date = models.DateField(auto_now_add=True)
```

```
    due_date = models.DateField()
    schedule_date = models.DateField()
    closed = models.BooleanField(default=False)

    def __unicode__(self):
        return self.name

    def colored_due_date(self):
        due_date = django_date(self.due_date, "d F Y")
        if self.due_date - timedelta(days=7) > date.today():
            color = "green"
        elif self.due_date + timedelta(days=7) > date.today():
            color = "orange"
        else:
            color = "red"
        return " <span style=color:%s>%s</span>" % (color, due_date)
    colored_due_date.allow_tags = True
```

Maintenant que vous avez compris le principe, nous vous laissons coder, à titre d'exercice, la méthode qui précise le nombre de jours restant avant le dépassement de la due-date.

Première conclusion

Remarquez comment, avec quelques lignes seulement, vous obtenez une interface d'administration totalement fonctionnelle. Vous avez vu comment modifier l'apparence de certaines parties de l'administration via soit le fichier `admin.py`, soit le fichier `models.py`. Dans la dernière partie, nous vous avons également montré la très grande souplesse de Django avec l'import de fonctions des templates directement dans les modèles. Normalement, vous commencez à entrevoir toutes les possibilités offertes par ce framework d'exception qu'est Django.

Dans un mode de développement par itérations, vous pourriez dès maintenant présenter votre projet à votre client. Il jouerait avec l'administration et validerait ou non l'architecture de vos modèles. Il pourrait donc donner son avis très tôt dans le développement de votre application ; c'est, par l'exemple, le principe d'un développement agile.

Il reste bien entendu quantité de choses à faire dans l'interface d'administration. Pourtant, laissons derrière nous cette partie pour le moment et occupons-nous désormais de l'interface client.

Gestionnaire de versions : révision 4

Retrouvez cette étape à la révision 4 du projet.

Seconde itération : les bases de l'interface client

Contrairement à l'interface d'administration, l'interface client est celle que verront les internautes se connectant à votre site. C'est là que réside le cœur de Django et c'est là que nous allons entrer de plain-pied dans la logique MVC.

Avant de créer vos premières vues, nous souhaitons vous faire découvrir une nouvelle application des *contribs*. Cette application `databrowse` vous aidera à mettre en place une interface utilisateur aussi simplement que vous avez mis en place l'interface d'administration.

> **Les contribs**
>
> Le framework Django se compose principalement de deux parties distinctes. La première, c'est le cœur de Django, dont nous allons parler tout au long de ce livre. C'est un ensemble à la fois monolithique et cohérent. La seconde est constituée d'un ensemble d'applications optionnelles, les *contribs*, qui activent au besoin certaines fonctionnalités.

L'application databrowse (contribs...)

`databrowse` est très proche dans son fonctionnement de l'`admin` de Django. Comme cette dernière, `databrowse` lit vos modèles pour vous offrir une interface utilisateur riche permettant de naviguer à travers vos objets (ici, les différents tickets). Pour mettre en place `databrowse`, il faut reprendre le même principe que pour n'importe quelle application : d'abord, ajouter l'application aux settings, puis « brancher » ses URL au fichier `urls.py` à la racine de votre projet.

Ouvrez le fichier `settings.py` et modifiez `INSTALLED_APPS` en ajoutant la nouvelle application : `django.contrib.databrowse`.

> **VERSION databrowse déprécié ?**
>
> À l'heure où nous écrivons ces lignes, la vie de `databrowse` est menacée. Cette application n'a malheureusement pas su trouver son public et il est maintenant question de la supprimer. Des projets sont dans les tiroirs pour en faire une option de l'`admin`. Dans le cas où `databrowse` ne serait plus disponible au moment où vous lirez ces lignes, vous pourrez utiliser https://bitbucket.org/boblefrag/django-prototype qui propose des fonctionnalités similaires.

Vous allez configurer `databrowse` de la même façon que vous avez créé le fichier `admin.py` pour configurer l'application `admin`. Il vous faut donc créer le fichier `ticket/databrowse_config.py` et l'éditer. Sa syntaxe est très proche de celle de l'`admin` :

```
from django.contrib import databrowse
from models import Task
databrowse.site.register(Task)
```

Ensuite, ouvrez le fichier `urls.py` à la racine de votre projet, ajoutez une URL qui pointera vers `databrowse` et importez le module `ticket.databrowse_config` :

```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
from django.contrib import admin, databrowse
from ticket.databrowse_config import *
admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'tracker.views.home', name='home'),
    # url(r'^tracker/', include('tracker.foo.urls')),
    url(r'^ticket/', include('ticket.urls')),
    url(r'^databrowse/(.*)', databrowse.site.root),
    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    url(r'^admin/', include(admin.site.urls)),
)
```

Ceci fait, il ne vous reste plus qu'à vous rendre à l'adresse `http://localhost:8000/databrowse` et à explorer votre nouvelle application.

Databrowse
Home / Tasks / une tâche en retard
Task: une tâche en retard
ID 2
Name une tâche en retard
Description Ceci est une tache en retard
Created date 19 décembre 2012
Due date 19 décembre 2012
Schedule date 19 décembre 2012
Closed No

Figure 2–17 Une vue de databrowse en fonctionnement

Comme vous le voyez, Django a mis en place un site complet vous permettant de naviguer librement dans vos objets. Vous pouvez filtrer vos tickets par dates de création, noms, etc. Il vous fournit même une vue calendrier pour chaque champ `date` de vos modèles. Sachez qu'il est possible de configurer `databrowse` plus finement encore.

Il est temps maintenant de réaliser vous-même vos propres vues.

Gestionnaire de versions : révision 5

Retrouvez cette étape à la révision 5 du projet.

Une première vue : liste

Pour cette première vue, nous vous proposons de lister sur une page tous les tickets. Cela servira de page d'accueil pour votre tracker, et vous permettra en un coup d'œil de savoir ce qu'il vous reste à faire et ce qui est le plus urgent.

La première chose à faire pour mener à bien cette tâche est de sélectionner les tickets qui vous intéressent. C'est assez simple ici puisque vous souhaitez récupérer tous les tickets. Mais peut-être voulez-vous les classer du plus urgent au moins urgent ? Voyons comment procéder avec Django.

Ouvrez le fichier `ticket/views.py` et importez-y le modèle `Ticket` :

Fichier ticket/views.py

```
from models import Task
```

Vous avez maintenant tout ce qu'il vous faut pour créer votre première vue :

```
def ticket_listing(request) :
    # Récupérez dans la base de données les éléments qui vous intéressent
    objets = Task.objects.all().order_by('-due_date')
```

Introduction aux querysets

Arrêtons-nous quelques instants sur cette dernière ligne de code. Vous avez déjà vu la syntaxe `Ticket.objects` quand vous avez utilisé la commande `python manage.py shell`. Il s'agit d'un *manager*. Les managers génèrent des *querysets* qui sont les points d'entrée principaux vers l'ORM de Django. C'est à travers ces eux que vous allez récupérer en base de données les objets que vous allez ensuite traiter puis présenter à l'utilisateur.

Le champ d'utilisation des managers est très large, mais voyons dès maintenant quelques fonctions très utiles.

- `all()` donne accès à tous les objets de votre modèle.
- `filter()` filtre les objets sur un champ, par exemple pour ne récupérer que les objets dont le champ `closed` est à `True`).
- `order_by()` classe les résultats selon un champ. Le signe - inverse le classement.

Présenter l'information : les templates

Maintenant que vous avez vos objets, vous devez choisir la façon dont ils vont être présentés à l'utilisateur, les champs qui seront affichés, etc. Tout ceci est du domaine du *template*.

Le principe est assez simple : vous avez un template destiné à accueillir des données et ces dernières sont présentées sous la forme d'un dictionnaire : le `Context`. Le template et le contexte sont compilés par Django dans une page HTML qui est ensuite présentée à l'utilisateur. Voici comment cela se présente de la façon la plus simple.

```
def ticket_listing(request)
    from django.template import Template, Context
    # Tout d'abord on définit un template :
    objets = Task.objects.all().order_by('-due_date')
    template = Template('{%for elem in objets %} {{elem}}<br /> {%endfor%}') ❶
    context = Context({'objets': objets})
    return HttpResponse(template.render(context))
```

Comme c'est la première fois que vous êtes confronté au langage de template de Django, regardons un peu plus en détail la ligne ❶.

- `{% for elem in objets %}` : il s'agit d'un *template tag* qui prend toujours la syntaxe `{% templatetag %}`. En l'occurrence, il est très similaire à une boucle for Python classique. Il est en deux parties : `{% for elem in objets %}` et `{%endfor%}` ; le code entre les deux se verra exécuter autant de fois qu'il y a d'objets dans la variable `objets`.
- `{{elem}}` : c'est une variable qui affiche l'objet `elem`. Ici, nous ne précisons pas d'attribut spécifique. `elem` sera donc le `name` de l'objet `ticket`, car vous avez déjà défini la méthode `__unicode__` de ce modèle dans la classe `Ticket`. Si vous aviez voulu accéder à la due-date de l'objet, vous auriez utilisé la syntaxe `{{elem.due_date}}`. Il en va de même pour les autres champs.

Pour tester votre toute nouvelle vue, il faut la déclarer dans vos URL :

```
from django.conf.urls.defaults import patterns, url
from views import home, ticket_listing

urlpatterns = patterns('',
    url(r'^home/$', home, name="home"),
    url(r'^home/(\w+)$', home, name="home"),
    url(r'^$', ticket_listing, name="listing")
                    )
```

Rendez-vous donc sur la page `http://localhost:8000/ticket/` (voir figure 2-18).

Figure 2–18
Une première liste un peu fruste

Certes, c'est un premier résultat, mais ce n'est pas vraiment ce qu'on attend d'un framework comme Django – ce n'est pas vraiment DRY. Vous auriez sans doute envie de présenter plus de champs, de faire une page web plus attractive, et de ne plus mélanger l'affichage et la logique dans le même fichier. Voyons donc dès maintenant comment rendre cet affichage plus agréable aussi bien pour l'utilisateur que pour le développeur.

Gestionnaire de versions : révision 6

Retrouvez cette étape à la révision 6 du projet.

Les fichiers templates

Pour rendre l'affichage plus convivial et plus DRY, nous vous proposons d'utiliser des fichiers qui vont contenir vos templates. Ils reprendront le langage de template qui est un mélange de HTML et de code spécifique à Django. Pour ce faire, vous allez devoir indiquer à Django où trouver les templates, dans le fichier `settings.py` de votre application.

Éditez la variable `TEMPLATE_DIRS` en indiquant le chemin complet vers vos templates. Nous vous recommandons de placer ces derniers dans le dossier `tracker` en changeant `/home/user` par le chemin absolu vers votre projet, comme suit :

```
TEMPLATE_DIRS = (
    '/home/user/tracker/templates',
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/
templates".
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
)
```

Créez un dossier `templates` à la racine de votre dossier `tracker`, puis un sous-dossier du nom de votre application, ici `ticket`. Ainsi, tous vos templates seront classés par applications ; il sera ainsi plus facile pour vous de vous y retrouver.

Dans le sous-dossier `ticket`, vous pouvez alors créer votre premier template, par exemple `list.html` :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Application Tracker</title>
    <meta http-equiv="content-type" content="text/html;charset=utf-8" />
   </head>
   <body>
    <h1>Liste des tickets</h1>
    {% for elem in objets %}
   <h3>
   {% if elem.closed %}
    <s>
    {{elem}}
    </s>
    {% else %}
      {{elem}}
    {% endif %}
   </h3>
   <p>{{elem.description}}</p>
   {% autoescape off %}
   <span>Due date :{{elem.colored_due_date}}</span>
   <span>Schedule date {{elem.schedule_date}}</span>
   {% endautoescape %}
   {% endfor %}
   </body>
</html>
```

La seule partie de ce code que vous ne connaissez pas est celle mise en exergue, mais elle s'explique d'elle-même. Il s'agit une fois encore d'un template tag. Selon que la variable `closed` est vraie ou fausse, le titre du ticket sera ou non barré.

Nous avons également utilisé le template tag `autoescape`. En effet, pour des raisons de sécurité, Django échappe par défaut les caractères HTML pour qu'ils ne soient pas rendus par le template. Ici, nous souhaitons utiliser `colored_due_date` ; nous désactivons donc l'échappement automatique pour cette partie du code.

Un raccourci : render_to_response

Il reste encore à éditer votre vue pour indiquer à Django d'utiliser le nouveau template que vous venez d'écrire. Pour cela, vous allez vous servir de l'un des nombreux raccourcis que Django met à votre disposition : `render_to_response`.

La syntaxe de `render_to_response` est la suivante :

```
render_to_response(template, contexte)
```

Vous pouvez donc modifier votre vue comme ceci :

```
from models import Task
from django.shortcuts import render_to_response
def ticket_listing(request) :
    objets = Task.objects.all().order_by('-due_date')
    return render_to_response('ticket/list.html', {'objets' : objets})
```

Rechargez votre navigateur pour constater que l'affichage est désormais bien plus convivial (voir figure 2-19).

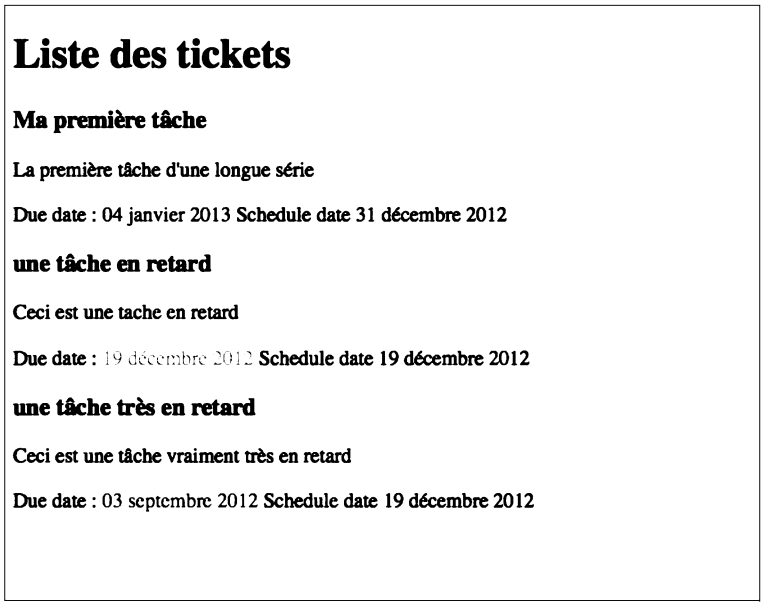

Liste des tickets
Ma première tâche
La première tâche d'une longue série
Due date : 04 janvier 2013 Schedule date 31 décembre 2012
une tâche en retard
Ceci est une tache en retard
Due date : 19 décembre 2012 Schedule date 19 décembre 2012
une tâche très en retard
Ceci est une tâche vraiment très en retard
Due date : 03 septembre 2012 Schedule date 19 décembre 2012

Figure 2–19
Le listing des tickets

Gestionnaire de versions : révision 7

Retrouvez cette étape à la révision 7 du projet.

Troisième itération : l'interface utilisateur, aspects avancés

Maintenant que vous savez comment trouver l'information dans la base de données et l'afficher, nous allons aborder certains aspects plus avancés de Django. Vous saurez afficher et trier vos tickets de la manière qui vous convient et serez en mesure d'utiliser votre tracker.

Mise en place des URL

La première vue étant écrite, vous pouvez prendre un peu de hauteur sur votre projet et imaginer les URL qui vous seront utiles.

- La liste de tous les tickets : c'est la vue que vous venez d'écrire.
- La liste de tous les tickets ouverts, pour déterminer les tâches qu'il vous reste à accomplir.
- La liste de tous les tickets fermés, pour estimer le travail déjà accompli.
- La liste des tickets dont la due-date est à moins d'une semaine.
- Une URL pointant sur un ticket précis.

À l'aide de Django, cela se fait très simplement avec seulement deux vues et deux URL. En définitive, vos URL se divisent en deux catégories : des listes d'objets et une vue de détail d'un objet.

Vous pouvez écrire vos URL de listes comme ceci :

- `http://localhost:8000/tracker/list/<filtre>/`
- `http://localhost:8000/tracker/detail/<id>/`

Voici comment les représenter dans votre fichier `ticket/urls.py` :

```
url(r'^list/(\w+)',ticket_list,name='list'),
url(r'^detail/(\d+)', ticket_detail, name='detail'),
```

Voyez comme vos URL sont très proches dans leur syntaxe de la fonction `home` que vous aviez écrite en début de chapitre.

N'oubliez pas d'importer les fonctions `ticket_list` et `ticket_detail` dans vos URL. Elles vont recevoir respectivement un paramètre de type `string` et `integer`.

La fonction `ticket_detail` est très simple. Elle prend un paramètre qui correspond à l'identifiant numérique unique que Django attribue automatiquement à vos objets `Ticket`. Il vous suffit donc de récupérer l'objet en question dans la base de données.

```
def ticket_detail(request, id):
    objet = Task.objects.get(pk=id)
    return render_to_response('ticket/detail.html', {'objet': objet})
```

Pour récupérer un seul objet dans la base de données, il faut une nouvelle méthode du manager du modèle `Ticket` : `get()`. Cette dernière s'attend à récupérer un objet unique ; par conséquent, elle vous renverra une erreur si la base de données vous retourne plus d'un objet. En paramètre de cette méthode, nous utilisons ici le raccourci `pk` pour `primary key` (clé primaire). Il s'agit de l'identifiant numérique unique que Django crée pour vous à chaque nouvel objet. L'employer vous assure qu'un seul objet sera retourné par la base de données.

Le template `ticket/detail.html` pourra dans sa forme la plus simple se présenter comme suit :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Application Tracker detail</title>
    <meta http-equiv="content-type" content="text/html;charset=utf-8" />
   </head>
   <body>
   <h3>
   {% if objet.closed %}
    <s>
    {{objet}}
    </s>
    {% else %}
      {{objet}}
    {% endif %}
   </h3>
   <p>{{objet.description}}</p>
    {% autoescape off %}
   <span>Due date :{{objet.colored_due_date}}</span>
   <span>Schedule date {{objet.schedule_date}}</span>
    {% endautoescape %}

   </body>
</html>
```

Pour la fonction `ticket_list`, une première idée pourrait être de la définir ainsi :

```
from datetime import date, timedelta
def ticket_list(request, filter):
    if filter == 'closed':
        objets = Task.objects.filter(closed=True)
    elif filter == 'open':
        objets = Task.objects.filter(closed=False)
```

```
    elif filter == 'week':
        objets = Task.objects.filter(
            due_date__lte=date.today() + timedelta(days=7)).filter(
                due_date__gte=date.today())
    return render_to_response('ticket/list.html', {'objets': objets})
```

Vous pouvez tester cette fonction et constater qu'elle est parfaitement opérationnelle. Au passage, remarquez l'usage de `__gte` et `__lte`.

- `__gte` renverra les objets dont l'attribut est égal ou plus grand que la valeur de test.
- `__lte` renverra a contrario les objets dont l'attribut est égal ou plus petit que la valeur de test.

Par ailleurs, les filtres peuvent s'enchaîner aussi souvent que nécessaire :

```
objets = Task.objects.filter( due_date__lte=date.today() +
timedelta(days=7)).filter(due_date__gte=date.today())
```

Gestionnaire de versions : révision 8

Retrouvez cette étape à la révision 8 du projet.

Cette fonction, toute opérationnelle qu'elle soit, souffre néanmoins de plusieurs défauts : beaucoup de `if`/`elif` en cascade, des répétitions et un code qui ne sera pas extensible.

Utiliser le dictionnaire GET

Pour adjoindre de nouveaux filtres, il vous faudra ajouter de nouveaux `if`, ce qui alourdira rapidement votre fonction et la rendra difficile à maintenir. Pour corriger ces défauts, vous allez profiter de l'objet `request`, dictionnaire qui contient quantité d'objets relatifs à la requête du navigateur du client que vous avez vu en début de chapitre. Entre autres attributs, il contient le paramètre `GET`. Ce dernier est lui aussi un dictionnaire contenant un ensemble de clés/valeurs.

Une URL du type `http://localhost:8000/tracker/list?open=false` enverra à Django un objet `request` contenant le dictionnaire `GET` suivant.

```
request.GET = {'open' :'false'}
```

Il est donc tout à fait possible d'utiliser ce dictionnaire pour simplifier drastiquement les URL et surtout la vue `ticket_list`.

```
url(r'^list/(\w+)', ticket_list, name='list'),
```

deviendra :

```
url('^list/$', ticket_list, name='list'),
```

Et la fonction `ticket_list` peut désormais s'écrire de cette manière :

```
def ticket_list(request):
    format = "%Y-%m-%d"
    objets = Task.objects.all()
    if request.method == 'GET':
        if 'closed' in request.GET:
            if request.GET['closed'] == "true":
                objets = objets.filter(closed=True)
            else:
                objets = objets.filter(closed=False)
                if 'start' in request.GET and request.GET['start'] != '':
                    objets = objets.filter(
                        due_date__gte=datetime.strptime(
                            request.GET['start'], format)
                        )
                if 'end' in request.GET and request.GET['end'] != '':
                    objets = objets.filter(
                        due_date__lte=datetime.strptime(
                            request.GET['end'], format)
                        )
    return render_to_response('ticket/list.html', {'objets': objets})
```

Les avantages d'une telle écriture sont multiples.

- Vous n'avez plus qu'une URL pour les listes. Que ce soit pour lister tous les objets `Task` ou une partie seulement, vous utiliserez toujours l'URL `http://localhost:8000/tracker/list`.
- Il vous sera toujours possible d'ajouter des paramètres au fur et à mesure de l'évolution de votre application.
- Vous venez de gagner beaucoup en termes de souplesse. En laissant l'utilisateur sélectionner la date de début et la date de fin, vous pouvez lister tous les tickets dont la due-date est fixée entre ces deux dates.
- Les différents filtres sont cumulatifs, ce qui vous autorise d'ores et déjà à réaliser des recherches assez détaillées. Par exemple, pour lister tous les tickets fermés dont la due_date est fixée entre le 2 octobre 2011 et le 3 juin 2012, l'URL pourra être :

```
http://localhost:8000/tracker/list?start=2011-10-2&end=2012-6-3&closed=true
```

C'est à la fois compréhensible, simple, élégant et pratique pour l'utilisateur, qui pourra aisément copier-coller cette URL pour la partager, y revenir ultérieurement, etc.

Bien sûr, pour rendre cette fonctionnalité disponible pour vos utilisateurs, vous avez besoin de créer un formulaire de recherche dans votre template. Cela ne présente aucune difficulté particulière puisqu'il s'agit tout simplement d'un formulaire HTML classique :

```
<form method='GET' action = ''>
  <label>Commencé depuis :</label><input type='text' name='start' />
  <label>Fini avant :</lable><input type='text' name='end' />
  <label>Fermé ?</label><input type='checkbox' name='closed' value='true'/>
  <input type="submit" value="filtrer" />
</form>
```

Nous avons ici proposé un type `text` pour les champs `start` et `end`. Un menu déroulant serait sans doute plus indiqué, mais comme nous allons modifier ce formulaire dans un instant, ne perdons pas de temps avec ce détail.

Gestionnaire de versions : révision 9

Retrouvez cette étape à la révision 9 du projet.

Vous remarquerez aussi que si l'un des champs est mal formaté, Django vous renverra une erreur. C'est un point sur lequel nous reviendrons en détail lorsque nous aborderons les formulaires et, en particulier, la partie « validation des données ».

Pour l'heure, nous allons tester les fonctionnalités déjà implémentées de votre application et utiliser un calendrier JavaScript pour la rendre plus conviviale. Ce sera également l'occasion d'aborder les médias, point très important dans une application Django.

Les fichiers statiques

Les fichiers statiques sont les images, les feuilles de style et les scripts JavaScript que vous souhaitez ajouter à vos pages. Lors de leur configuration, il vous faut garder une chose à l'esprit : le développement n'est pas la production. Durant la phase de développement, vous pouvez tout à fait laisser Django servir vos fichiers statiques. Le but de cette phase n'étant pas la recherche de la performance, cela ne pose aucune difficulté. En revanche, dans le cadre de la mise en production, ne laissez jamais Django servir vos fichiers statiques, car cela aurait des conséquences calamiteuses sur la performance de votre application. En effet, chaque fichier statique que Django va servir sera directement chargé en mémoire ; or, les serveurs web actuels sont extrêmement performants pour servir des fichiers statiques. Nous vous expliquerons en détail, lors de la mise en production de votre application, comment configurer votre serveur web

pour qu'il serve les fichiers statiques en lieu et place de Django. Pour le moment, voyons comment configurer simplement les fichiers statiques lors de la phase de développement.

Comme vous vous en doutez, c'est une nouvelle fois le fichier `settings.py` à la racine de votre application qu'il va falloir éditer.

- `STATIC_ROOT` : cette variable d'environnement indique à Django où se trouve la racine de vos fichiers statiques. Vous devez utiliser ici un chemin complet, par exemple `/home/user/tracker/static`.
- `STATIC_URL` : cette variable d'environnement indique à Django sur quelle URL servir les fichiers statiques. Il s'agit d'un préfixé, donc si vous indiquez `'/static/'`, vos fichiers statiques seront disponibles sous l'URL `http://localhost:8000/static/`.

Il ne vous reste plus qu'à créer le dossier `static` et à y déposer vos fichiers statiques.

staticfiles

`staticfiles` est une application très pratique pour le développement de votre application ; elle vous permet, dans un premier temps, de ne pas trop vous soucier des fichiers statiques.

Intégrée dans Django, elle va vous servir à deux niveaux. Tout d'abord, durant le développement de votre application, Django va servir les fichiers présents dans le dossier `static` de toutes vos applications. Ensuite, lorsque vous souhaiterez passer en production, il vous faudra retrouver vos fichiers statiques éparpillés entre les différentes applications et les regrouper en un seul dossier de manière à les rendre disponibles pour votre serveur (Apache, Ngnix...). Pour cela, la commande `python manage.py collectstatic` va copier pour vous le contenu du dossier `static` de chacune de vos applications dans le dossier correspondant à `STATIC_ROOT`. Il vous suffira alors de créer une règle dans la configuration de votre serveur pour servir correctement vos fichiers statiques en production.

STATIC_URL et render_to_response

Bien entendu, il n'est pas question d'indiquer « en dur » dans vos templates le chemin des fichiers statiques car cela n'est pas DRY : si vous décidez de déplacer vos fichiers statiques, c'est dans tous vos templates que vous devrez faire la modification ! Django fournit donc la variable `{{STATIC_URL}}` que vous utiliserez pour indiquer où se trouvent vos fichiers statiques. Par exemple, pour charger sur votre page d'accueil le logo `tracker/ticket/static/logo.png`, vous utiliserez la syntaxe suivante :

```
{{STATIC_URL}}/logo.png
```

Si vous êtes en développement, Django va chercher `logo.png` dans chaque dossier `static` de vos applications et le servir sous l'URL `/static/logo.png`. Si vous êtes en production, c'est-à-dire si votre application est servie par un autre serveur web que Django (Apache, Ngnix…), vous devez créer une règle dans la configuration de votre serveur pour faire pointer l'URL `/static/` vers votre dossier `STATIC_ROOT`. La commande `python manage.py collectstatic` se chargera de copier votre fichier `logo.png` dans le dossier qui convient.

L'avantage de cette méthode réside dans le fait que si vous souhaitez changer l'emplacement de vos fichiers statiques ou l'URL qui les sert, vous n'aurez à modifier que `STATIC_ROOT` ou `STATIC_URL` dans le fichier `settings.py`.

Seulement, si vous testez cette variable dès maintenant vous vous apercevrez que cela ne fonctionne pas. En effet, le contexte que vous passez à vos templates ne comprend pas la variable `{{STATIC_URL}}`.

RequestContext

Remplir le contexte avec des objets que vous n'avez pas besoin de remplir à chaque fois, c'est le but des `TEMPLATE_CONTEXT_PROCESSORS`. Django vous fournit de base ceux dont vous avez besoin pour commencer. Sachez que vous avez la possibilité de configurer et de choisir ceux que vous souhaitez utiliser, et vous pouvez bien entendu en écrire vous-même, pour vos propres besoins.

Il suffit de charger les `TEMPLATE_CONTEXT_PROCESSORS` dans vos vues. Pour cela, une fois encore, Django vous fournit une aide précieuse via la classe `RequestContext`, héritant de `Context` que vous avez déjà manipulée. `RequestContext` prend en argument l'objet `request` et rend disponible dans les templates un certain nombre d'objets prédéfinis, dont `{{STATIC_URL}}`.

Reprenant l'exemple de la vue `ticket_list`, voici comment la modifier pour utiliser `RequestContext` :

```
from django.template import RequestContext
return render_to_response('ticket/list.html',
                          {'objets': objets},
                          RequestContext(request)
                          )
```

Gestionnaire de versions : révision 10

Retrouvez cette étape à la révision 10 du projet.

Au premier abord, il peut sembler fastidieux de devoir ajouter `RequestContext(request)` à chaque fois que vous souhaitez rendre une vue, mais

n'oubliez pas que lorsque vous l'utilisez, vous effectuez quelques requêtes supplémentaires en base de données. Django vous laisse donc le choix de faire ou non ces quelques requêtes supplémentaires.

Render

Il existe un raccourci Django qui rend votre vue avec une syntaxe plus concise. `render` s'utilise de la même manière que `render_to_response`, mais il instancie un objet `RequestContext` sans qu'il y ait besoin de le préciser. Ainsi :

```
return render_to_response('ticket/list.html',
                          {'objets': objets},
                          RequestContext(request)
                          )
```

et

```
return render(request, 'ticket/list.html',
                          {'objets': objets},
                          )
```

sont équivalents.

Pour utiliser `render`, vous devez bien entendu commencer par l'importer :

```
from django.shortcuts import render
```

Une petite astuce

Comme vous l'avez vu jusqu'ici, Django suit en tout point la logique DRY, aussi bien dans les URL que dans les vues ou dans l'interface d'administration. En tout point ? Pas forcément. Vous avez peut-être été surpris de voir que dans `settings.py`, les variables de templates ou de fichiers statiques devaient être des chemins absolus. Que faire si l'on souhaite déplacer son projet ou quand un autre développeur désire installer l'application sur sa machine ? Voici une petite astuce pour faire en sorte que votre projet reste DRY.

Au tout début de votre fichier `settings.py`, écrivez :

```
import os
project_path = os.path.dirname(os.path.abspath(__file__))
```

La variable `project_path` va ainsi prendre la valeur du chemin complet vers le dossier de votre projet.

Ensuite, pour STATIC_ROOT, par exemple, vous pouvez écrire :

```
STATIC_ROOT = project_path + '/static/'
```

Maintenant, vous pouvez facilement déplacer votre projet dans l'arborescence de votre ordinateur ou l'installer sur une autre machine sans avoir besoin de vous soucier des settings.

Conclusion

Au cours de cette première découverte du framework Django, vous venez de créer votre première application. Il vous faudra sans doute un peu de temps pour maîtriser totalement le cheminement de développement d'une application. Pourtant, vous avez pu remarquer qu'en quelques commandes simples et quelques lignes de code, Django est en mesure de vous proposer une architecture modulable et claire.

L'un des grands atouts d'un framework tel que Django réside dans ces aspects de simplicité et de pragmatisme. À mesure que vous apprendrez à le maîtriser, vous découvrirez de nouvelles fonctionnalités, de nouvelles méthodes de travail et, surtout, à quel point Django a été construit autour d'une maxime bien connue : « rendre faciles les choses difficiles et rendre faisable l'impossible. »

3

Des bases à la production : créer un agenda partagé

Le tracker que vous venez de réaliser n'était qu'un prétexte pour vous apprendre les grands principes de Django. Maintenant que vous avez découvert les bases du framework, vous êtes fin prêt à rédiger une application complète : un agenda partagé. Pour ce faire, vous allez bien entendu réutiliser une partie de ce que vous avez découvert précédemment.

Un agenda partagé n'est pas si différent de l'application que vous venez de créer, car c'est aussi un outil de gestion du temps. Cependant, là où un tracker est destiné plutôt à des développeurs de logiciels, l'agenda s'adresse à tous. Vous n'allez pas pour autant abandonner le tracker que vous avez conçu. Bien au contraire, au cours de l'évolution de l'agenda partagé, nous en continuerons le développement à titre d'exercice. Nous vous conseillons également de l'utiliser pour vous-même, d'ajouter des tâches, de les ouvrir et de les fermer, et de suivre l'évolution de votre travail. Outre les intérêts d'une telle technique que nous avons déjà abordés, cela vous permettra de faire du *dog fooding*.

> JARGON **Dog fooding**
>
> Le *dog fooding* est une expression qui se réfère au fait de goûter soi-même ce qu'on donne à manger à son chien. Au sens figuré, le plus souvent dans le cadre du développement logiciel, il s'agit d'éprouver soi-même les outils que l'on a créés en les utilisant, afin d'en confirmer les mérites en termes d'efficacité, de richesse, etc.

C'est bien souvent l'usage qui détermine les fonctionnalités d'un logiciel. En employant le tracker que vous avez créé, vous déterminerez aisément les fonctionnalités qui vous manquent et serez donc en mesure de les développer, de les tester par l'utilisation, et ainsi de suite. Après de longues heures d'utilisation, vous aurez à votre disposition un tracker parfaitement adapté à vos besoins et un agenda partagé que vos amis auront plaisir à utiliser.

Fonctionnalités de l'application

Ajoutons aux bases d'un tracker quelques principes tirés d'un réseau social, un peu de mise en forme... et nous obtiendrons un agenda partagé ! Un utilisateur crée un agenda, dans lequel il enregistre des événements et invite d'autres utilisateurs. Il peut alors gérer les participants, consulter les événement auxquels il a été invité, les réponses qu'il a données ou reçues, etc. Nous traiterons donc dans cette section la gestion d'un calendrier et des événements, le flux des invitations, etc.

Cette deuxième application, comme la première, n'est bien entendu qu'un prétexte pour apprendre à maîtriser les concepts fondamentaux de Django. Néanmoins, l'objectif avoué est qu'à la fin, vous ayez entre les mains une application parfaitement fonctionnelle. À chaque étape, vous pourrez comparer votre travail aux exemples de code présents sur https://bitbucket.org/boblefrag/livre_django_agenda. Nous vous indiquerons, avant chaque extrait de code, la révision à laquelle vous devez vous rendre pour obtenir la correction, de la manière suivante :

rev 65

Création de la base de données.

Comme cela est précisé dans le titre de ce chapitre, cette application va être développée jusqu'à sa mise en production. Ainsi, le premier démarrage sera un peu différent de ce que vous avez appris lors de la mise en place du tracker.

Un environnement complet

Pour cette application, nous proposons de mettre en place un environnement de travail complet, c'est-à-dire que vous allez utiliser tous les outils à votre disposition pour vous faciliter la tâche tout au long du développement. Vous utiliserez les deux applications `virtualenv` et `pip`, déjà employées lors de la création du tracker.

ALTERNATIVE apt et chroot sous Debian

Les outils dont nous allons parler sont aujourd'hui les standards de fait en développement Django. Toutefois, vous pouvez tout à fait vous en passer pour développer des applications de qualité, en particulier si vous êtes sous un environnement GNU/Linux, où des outils équivalents sont déjà disponibles au niveau du système lui-même. Par exemple, sous GNU/Linux/Debian, environnement de choix pour le développement de logiciels du fait de sa très grande stabilité et de son développement communautaire, `pip` sera avantageusement remplacé par `apt` et `virtualenv` par `chroot`.

Les paquets Django externes que nous installerons sont tous disponibles dans les dépôts des distributions Linux majeures. Ainsi, quand vous rencontrerez la commande :

```
pip install south
```

vous pourrez la remplacer, sous Debian par exemple, par la commande :

```
sudo apt-get install django-south
```

Les outils que nous vous proposons de découvrir vous seront donc particulièrement utiles dans le cas où vous travaillez dans un environnement Windows ou Mac OS X, où un gestionnaire de paquets n'est pas disponible, ou dans le cas où vous travaillez de concert avec des développeurs externes dont c'est l'environnement.

Pour commencer ce nouveau projet, lancez la commande suivante :

```
virtualenv agenda_env
```

Activer l'environnement virtuel

Maintenant, vous avez votre environnement virtuel, mais il faut encore l'activer ; `virtualenv` vous simplifie le travail car il crée dans chaque environnement virtuel un script destiné à l'activer. Sous Windows, exécutez le fichier `activate.bat` du dossier `Scripts`. Sous Linux et Mac OS X, utilisez la commande suivante :

```
[user@localhost]$ source agenda_env/bin/activate
```

Vous pouvez vérifier l'environnement dans lequel vous êtes en regardant l'invite de commande de votre terminal :

```
(agenda_env)[user@localhost]
```

Vous devrez lancer cette commande à chaque fois que vous souhaiterez travailler sur votre projet. Pour désactiver l'environnement de travail, il suffit d'utiliser la commande `deactivate` :

```
(agenda_env)[user@localhost] deactivate
```

Vous pouvez également fermer tout simplement votre session ou activer un autre environnement de travail, ce qui aura le même effet.

Installer les paquets de base

Maintenant que votre environnement virtuel est créé et activé, il vous faut quelques paquets de base. Nous commencerons par le plus évident : Django lui-même ! Il est inutile de l'installer par la méthode classique disponible sur votre système car, comme nous venons de préciser à `virtualenv` de ne pas prendre en compte les paquets du système, cela n'aurait aucun effet sur le projet en cours.

Utilisez l'application `pip` (voir chapitre 1). Vérifiez par deux fois que votre environnement virtuel est bien activé et lancez la commande suivante :

```
pip install Django
```

Créer un fichier requirements.txt

La prochaine étape consiste à établir la liste de tous les paquets dont nous aurons besoin pour faire fonctionner notre application et à les enregistrer dans un fichier `requirements.txt`. Ce dernier pourra ensuite être utilisé, par exemple, par un autre développeur qui voudrait faire fonctionner votre application sur sa propre machine, ou tout simplement par vous-même lors de la mise en production.

Pour créer ce fichier, tapez :

```
pip freeze > requirements.txt
```

OUTIL Désinstaller et mettre à jour un paquet avec pip

Comme vous venez de le voir, `pip` est un outil de gestion des paquets très simple et pratique. Bien que ce ne soit pas le sujet de ce livre, voici deux commandes très utiles que `pip` met à votre disposition :

- `pip uninstall paquet` pour supprimer un paquet de votre environnement virtuel ;
- `pip upgrade paquet` pour mettre à jour un paquet à la version la plus récente disponible sur le site Internet PiPY.

▸ https://pypi.python.org/pypi

Configuration du projet

La configuration de l'agenda n'est pas différente de ce que nous avons vu pour le tracker. Initialisez votre projet via la commande suivante :

```
$ django-admin.py startproject agenda
```

Dirigez-vous maintenant dans le dossier de votre projet et créez votre première application :

```
$ cd agenda
$ python manage.py startapp personal_calendar
```

Prenez toujours bien soin de suivre votre projet via un gestionnaire de versions :

```
$ hg init
```

À la racine de votre dossier `agenda`, créez le fichier `.hgignore`, qui a pour fonction d'exclure certains fichiers de votre gestionnaire de versions. Ensuite, éditez-le et modifiez-le pour ignorer :

- les fichiers Python compilés ;
- les fichiers temporaires ;
- les fichiers de base de données.

```
#use global syntax
syntax:glob
*.pyc
*~
*.db
```

Dans le dossier `.hg`, créez le fichier `hgrc`, qui détermine quelques variables d'environnement pour votre projet. Par exemple, pour ne pas avoir à indiquer votre nom d'utilisateur, remplissez-le comme ceci :

```
[ui]
username = PrenomNom <prenom.nom@example.net>
verbose = True
```

Bien d'autres options sont disponibles dans ce fichier. Par exemple, il est possible d'y indiquer son éditeur de texte préféré, l'outil montrant les différences entre plusieurs versions ou branches d'un même projet… Pour tout cela, nous vous renvoyons à la documentation en ligne de Mercurial.

Si vous ne connaissez pas du tout Mercurial, la lecture du tutoriel Mercurial sur le site www.hginit.com est particulièrement recommandé. Bien qu'en anglais, sa lecture reste simple et amusante – vous apprendrez les bases de Mercurial autour d'une recette de guacamole un peu spéciale…

Ajoutez ensuite les fichiers de votre répertoire au gestionnaire de versions et procédez à votre premier `commit` :

```
$ hg add
$ hg commit
```

rev 0

Mon premier `commit`.

Éditez le `settings.py` de votre projet :

```
# settings.py
# Django settings for agenda project.
import os
project_path = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

DEBUG = True
TEMPLATE_DEBUG = DEBUG

ADMINS = (
     ('Prenom Nom', 'votre email'),
)

MANAGERS = ADMINS

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'devel.db',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    }
}
TIME_ZONE = 'Europe/Paris'
LANGUAGE_CODE = 'fr-fr'
SITE_ID = 1
USE_I18N = True
USE_L10N = True
MEDIA_ROOT = os.path.join(project_path, 'media')
```

```
MEDIA_URL = '/media/'
STATIC_ROOT = os.path.join(project_path,'static')
STATIC_URL = '/static/'
ADMIN_MEDIA_PREFIX = '/static/admin/'
STATICFILES_DIRS = (
)
STATICFILES_FINDERS = (
    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',
    'django.contrib.staticfiles.finders.DefaultStorageFinder',
)
SECRET_KEY = '4q#27@sgd-1!99!a3h)k4^rlu(&bct-brcgo$-gy^mnv4$7daf'
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
#     'django.template.loaders.eggs.Loader',
)

MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
)

ROOT_URLCONF = 'agenda.urls'

TEMPLATE_DIRS = (
    project_path + '/templates'
)

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
   # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    'django.contrib.admindocs',
    'south',
)
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'mail_admins': {
```

```
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler'
        }
    },
    'loggers': {
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': True,
        },
    }
}
try:
    from local_settings import *
except:

    pass
```

Plus d'efficacité avec south et local_settings de settings.py

Vous connaissez déjà le fichier `settings.py`. Notez que nous n'avons pas encore importé l'application `personal calendar`, puisque aucun modèle n'est encore défini dans cette application. Ce que vous ne connaissez pas, en revanche, c'est la partie qui concerne `south` dans les `INSTALLED_APPS` et l'instruction `from local_settings.py import *`.

Modifier le schéma de la base et garder trace des changements avec south

`south` est un paquet Python pour Django (on parle alors d'« application Django ») qui vous sera très utile. Il sert principalement à modifier le schéma de la base de données au fur et à mesure que vous développez votre application, sans devoir lancer de commandes SQL. Toutefois, les possibilités de `south` ne se limitent pas à cela ; cette application garde trace des changements effectués dans votre base de données, un peu à la manière d'un gestionnaire de versions, et vous permet de revenir en arrière tout au long du développement. Son utilisation reste très simple et s'intègre particulièrement bien aux commandes habituelles de Django.

Lorsque nous aurons besoin de créer notre base de données, nous reviendrons sur `south` et les commandes particulières qu'il propose. Pour le moment, commençons par l'installer :

```
pip install south
```

Ajoutons-le à notre fichier `requirements.txt` :

```
pip freeze > ../requirements.txt
```

Vous pouvez maintenant utiliser `syncdb` pour créer, dans votre base de données, les tables nécessaires aux différentes applications présentes dans `INSTALLED_APPS` :

```
python manage.py syncdb_
```

L'astuce from local_settings import *

Il s'agit ici d'une petite astuce très pratique quand vous travaillez avec plusieurs environnements. En l'occurrence, sur le projet d'agenda partagé, vous allez travailler sur votre machine et mettre en production sur une autre. Donc, les données relatives à la base de données, par exemple, ne seront pas les mêmes dans les deux environnements.

Si vous ne souhaitez pas devoir éditer le `settings.py` de votre application à chaque fois que vous mettez en production, `from local_settings import *` est fait pour vous ! Si vous devez travailler avec d'autres développeurs, vous voulez sans doute séparer de `settings.py` ce qui est valable quelle que soit la plate-forme (les applications installées, par exemple) et ce qui est spécifique à chaque développeur. Dans ce cas aussi, `from local_settings import *` est fait pour vous !

Voyons comment mettre en place cette astuce dans le cadre de deux environnements :

- l'un de développement ;
- l'autre de production.

Tout d'abord, créez le fichier `local_settings.py` dans le dossier `agenda` de votre projet. Il ne contiendra qu'une seule ligne :

```
from devel_settings import *
```

Créez ensuite, au même niveau, le fichier `devel_settings.py`. Indiquez dedans uniquement l'environnement de développement, comme la base de données, et le mode `DEBUG` :

```
DEBUG = True
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.sqlite3',
    'NAME': 'devel.db',
    'USER': '',
    'PASSWORD': '',
    'HOST': '',
    'PORT': '',
    }
}
```

Créez ensuite un fichier `prod_settings.py` dans lequel vous allez indiquer les mêmes informations, mais pour l'environnement de production. Par exemple :

```
DEBUG = False
DATABASES = {
  'default':
    {'ENGINE': 'django.db.backends.postgresql_psycopg2',
     'NAME': 'production_db',
     'USER': 'myuser',
     'PASSWORD': 'mypassword',
     'HOST': '',
     'PORT': '',
  }
}
```

Maintenant que tout est prêt, la « magie » va opérer. Vous allez suivre, à travers votre gestionnaire de versions, les fichiers `devel_settings.py` et `prod_settings.py`, mais pas le fichier `local_settings.py`. Donc, ajoutez ce dernier à votre `.hgignore` :

```
#use global syntax
syntax:glob
*.pyc
*~
*.db
agenda/local_settings.py
```

Ainsi, dans l'environnement de développement, `settings.py` importe `local_settings.py` qui lui-même importe `devel_settings`.

Une fois sur votre serveur de production, il vous suffira de créer le fichier `local_settings.py` et d'y écrire :

```
from prod_settings import *
```

Vous pouvez donc modifier le `settings.py` de production sur votre environnement de développement et être certain qu'il sera pris en compte au moment de la mise en production, sans jamais avoir à vous soucier d'autre chose.

Rev 1

Second `commit`. Installation de `south`, mise en place de l'astuce `from local_settings import *`.

Organisation du projet

La mise en place de votre projet est désormais terminée. Maintenant commence la création proprement dite. Vous allez concevoir vos modèles, c'est-à-dire l'organisation de la base de données. À ce stade, il est peu probable que vous ayez une vision parfaitement claire de la manière dont les modèles vont s'organiser et il est fort possible qu'au cours du développement de votre application, vous souhaitiez les modifier. C'est la raison pour laquelle nous vous avons fait installer `south`. Concevez vos modèles avec sérénité, sachant que vous pourrez à tout moment les modifier, ajouter des champs, en supprimer, ajouter des contraintes, etc.

Conception des modèles

Maintenant que tout est en place, vous allez pouvoir prendre quelques minutes de réflexion, idéalement loin de votre ordinateur, et commencer à imaginer à quoi doit ressembler votre application.

Deux approches différentes peuvent être adoptées :

- modéliser la table de routage (les URL) ;
- modéliser les modèles (la base de données).

Dans ce cas, il n'y a pas de bonnes ou de mauvaises pratiques ; c'est vraiment une affaire de goût. À vous de savoir avec quoi vous vous sentez le plus à l'aise.

Disons, pour simplifier, qu'écrire les URL avant les modèles, c'est penser l'application en commençant par l'interface alors que rédiger les modèles correspond plutôt à penser l'application d'un point de vue fonctionnalités. Ici, nous allons commencer par les modèles. Dans cette application, vous allez manipuler les objets suivants : des utilisateurs et des événements.

Avant de réfléchir plus avant, voyons comment ces deux modèles vont être reliés l'un à l'autre.

- Les utilisateurs vont avoir plusieurs événements.
- Les événements peuvent concerner plusieurs utilisateurs.

Dans ce cas, vous êtes clairement dans un contexte de relation « plusieurs vers plusieurs » (*many-to-many* en langage SQL).

Pour le moment, nous ne nous attarderons pas sur les utilisateurs. Django vous offre déjà un objet utilisateur tout à fait convenable. Dans un premier temps, vous allez donc utiliser ce modèle ; nous verrons plus tard comment l'enrichir très simplement. Maintenant, réfléchissons aux champs du modèle d'événement :

- un nom pour identifier facilement un événement ;
- une description (un champ texte plus étendu pour détailler l'événement) ;

- des participants ;
- une date ;
- un lieu.

Les participants sont ici les utilisateurs. Seulement, vous allez rapidement être confronté à un problème de taille. En effet, vous aimeriez sans doute pouvoir définir un statut particulier pour chaque participant, comme « invité », « confirmé », « désisté », etc. Or, ici, vous n'avez pas la possibilité d'ajouter une information supplémentaire sur votre relation. Heureusement, Django vous offre une fonctionnalité extrêmement pratique avec l'option `through` d'une relation many-to-many.

La relation many-to-many

Pour bien comprendre cette option, il faut savoir que, pour chaque relation many-to-many entre deux modèles, la base de données crée automatiquement une table intermédiaire chargée d'enregistrer toutes les associations existant entre les divers enregistrements de ces modèles. Ici, cette table de relation enregistre chaque couple utilisateur/événement existant.

L'option through

Avec l'option `through`, Django vous laisse la possibilité de déterminer quelle est la table de relation que vous souhaitez utiliser. Vous devez néanmoins créer deux champs de type `ForeignKey` pour lier vos tables `Event` et `User` de façon à ce que Django sache comment organiser la relation. Vous pourrez créer d'autres champs sur cette table, qui qualifieront plus finement cette relation.

Création des tables

Voici comment vont se présenter vos modèles :

```
# -*-coding:utf-8 -*-
from django.db import models
from django.contrib.auth.models import User
# On rend disponible l'objet User pour le référencer dans les modèles


class Evenement(models.Model):
    nom = models.CharField(max_length=250)
    description = models.TextField()
    participants = models.ManyToManyField(
                        User,
                        through="Evenement_Participant",
                    )
```

```
    date = models.DateTimeField()
    lieu = models.TextField()


class Evenement_Participant(models.Model):
    #notre table de relation
    evenement = models.ForeignKey(Evenement)
    participant = models.ForeignKey(User)
    status_choices = (
                      (0, "hôte"),
                      (1, "invité"),
                      (2, "désisté")
                      )
    status = models.IntegerField(choices=status_choices)
```

REMARQUE -*-coding:utf-8 -*-

Sans configuration particulière, Python utilise l'encodage de caractères ASCII qui n'accepte pas les caractères français comme les accents ou les cédilles.
L'instruction en début de fichier `#-*-coding:utf-8 -*-` indique à Python d'employer un jeu de caractères bien plus étendu. Ainsi, vous pouvez sans problème utiliser des caractères accentués dans votre fichier Python.

Votre fichier `models.py` est terminé.

Première migration avec south

Il est temps maintenant de tester vos modèles et de les utiliser, mais qu'adviendra-t-il si vous avez fait une erreur ou si vous changez d'avis ? Voici les solutions à votre disposition, mais aucune n'est vraiment satisfaisante :

- détruire entièrement votre base de données ;
- descendre au niveau de la base de données et faire les changements à la main.

L'application `south` nous ouvre une troisième voie : les migrations. Dès que vous modifiez vos modèles, `south` répercute le changement sur le schéma de la base de données, garde ces changements en mémoire et crée pour vous un script de régression.

Il faut maintenant indiquer à Django que vous venez de créer une nouvelle application. Pour cela, éditez votre fichier `settings.py` et ajoutez-la aux `INSTALLED_APPS` :

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
```

```
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    'django.contrib.admindocs',
    'south',
    'personal_calendar'
```

Lancez la première migration avec `south` :

```
python manage.py schemamigration personal_calendar --initial
```

Ensuite, comme vous y invite le retour de la ligne de commande, vous pouvez appliquer la migration avec :

```
python manage.py migrate personal_calendar
```

Plus tard, lorsque vous aurez besoin de modifier vos modèles, il vous suffira d'utiliser la commande suivante :

```
python manage.py schemamigration personal_calendar –auto
```

suivie de :

```
python manage.py migrate personal_calendar
```

Pour plus d'informations sur south et une documentation complète

- http://south.aeracode.org/

rev 2

Second `commit`, création des modèles et ajout de la première migration.

Test des modèles et TDD

Maintenant que vos modèles sont prêts, il est temps de les tester un peu. Bien entendu, il est possible de créer quelques objets et de vérifier leur comportement, grâce à la commande `manage.py shell`. Néanmoins, nous vous proposons de découvrir une technique bien plus efficace : le TDD (*Test Driven Developpement* ou développement dirigé par les tests). Cette technique va bien plus loin que le seul fait de tester son application : il s'agit de rédiger les tests avant de développer la fonctionnalité.

On peut y voir un workflow de développement de fonctionnalité. La première étape est toujours un instant de réflexion : quelle est la demande ? Quel est le comporte-

ment attendu ? Quelles sont les exceptions auxquelles il faut penser ? Une bonne idée pourrait être de consigner sur un papier toutes ces questions et leurs réponses. Seulement, au moment où vous allez développer votre fonctionnalité, peut-être aurez-vous égaré ce fameux papier ? Ou pire : plusieurs mois plus tard, vous reviendrez sur la fonctionnalité pour y ajouter une exception. Comment être certain que vous n'êtes pas en train de casser une autre partie de votre programme ?

Bon, direz-vous, il faudrait tenir à jour une liste des fonctionnalités et le comportement attendu de chacune. Le développement dirigé par les test, c'est cela et bien plus. Pendant la conception intellectuelle de votre application, vous allez rédiger le test de votre fonctionnalité, définir son comportement normal, tester la gestion des erreurs, etc. Ensuite, lorsque vous allez la coder, vous n'aurez plus qu'à vérifier que vos tests renvoient bien le résultat escompté.

Les tests dans les docstrings

Django est un framework Python ; vous manipulez donc des paquets (modules), des classes et des fonctions. En conséquence, tous les principes de Python s'appliquent aussi avec Django. L'un des concepts les plus puissants de Python reste les *docstrings* (*documentation strings* ou textes de documentation), qui sont un moyen très pratique d'insérer de la documentation dans le code source. Une docstring est une chaîne de caractères placée dans un attribut spécial `__doc__` en début de module, classe ou fonction. Cet outil est très utilisé en Python, car il permet de documenter le code source tout en le concevant.

> **OUTILS Documentation automatique**
>
> De nombreux outils créent automatiquement une documentation utilisateur via les docstrings. Citons, par exemple, l'application Python Sphinx, qui retourne une documentation PDF ou HTML, ou le paquet Django `django.contrib.admindoc`, qui offre une interface de documentation dans l'administration de Django.

Or, les docstrings peuvent également être utilisées comme des outils de test. On parle alors de *doctests*. Leur principe consiste à insérer dans les docstrings des extraits de code, comme ceux que l'on trouve dans l'interpréteur Python ou dans le `shell` de la commande `python manage.py shell`. Lorsque vous testerez un module, une classe ou une fonction, l'extrait de code présent dans les docstrings sera alors exécuté. Voici un exemple de docstring contenant un doctest :

```
"""
>>> print "Ok"
OK
"""
```

Comme vous le voyez, on indique une commande et le résultat attendu. Ce test vérifie que l'instruction `print "OK"` renvoie bien `"OK"`. Il y a de fortes chances que ce test réussisse ; en revanche, le test suivant échouera – en effet, en aucun cas l'instruction `print "OK"` ne renverra `"NOP"`.

```
"""
>>> print "OK"
NOP
"""
```

Pour bien comprendre ce que sont les docstrings, prenons un exemple plus parlant et complet :

```
def add(nombre1, nombre2):
    """
    cette fonction renvoie la somme de deux nombres donnés en argument.
    >>> add(5,2)
    7
    """
    return nombre1 + nombre2
```

L'avantage d'utiliser les docstrings pour rédiger des tests concis et simples est double.

- Vous testez votre application.
- Vous donnez un exemple d'utilisation de votre fonction dans votre documentation. Dès lors, un autre développeur, ou tout simplement vous-même dans quelques temps, pourra saisir au premier coup d'œil comment manipuler vos modèles. Nous ne vous recommandons pas de rédiger tous vos tests de cette façon, car cela alourdirait vos fichiers et vous perdriez l'avantage premier de faciliter la compréhension de votre code. La clé ici est de rester simple et sobre.

Ouvrez l'invite de commande Django et tapez chacune des instructions de l'exemple qui suit. Cela vous aidera à saisir comment écrire vos propres doctests. Nous vous conseillons d'ailleurs de toujours documenter vos modèles de la manière suivante.

1. Rédigez un court texte expliquant la fonction de votre modèle.
2. Écrivez une courte description de chacun des champs.
3. Ajoutez un test qui devrait contenir les éléments suivants :
   - la création d'un objet de la classe modèle que vous définissez ;
   - une ou deux courtes requêtes pour récupérer l'objet créé.

```
class Evenement(models.Model):
    """
    Cette classe renferme toutes les informations d'un événement.
```

```
  - un nom pour le retrouver facilement- une description pour enregistrer toutes
les informations annexes de cet événement - les participants, qui sont des
utilisateurs. Les informations supplémentaires sur les participants à cet
événement sont stockées sur le modèle Evenement_Participant. - la date à
laquelle se déroule l'événement
  - le lieu de l'événement (pour le moment, il s'agit uniquement d'un champ
texte, mais un vrai champ « géographique » serait sans doute plus utile).

    # Création d'un événement.
    # On importe datetime et timedelta pour manipuler les objets dates.
    # On importe également le modèle User de Django pour insérer un participant.
    >>> from datetime import datetime, timedelta
    >>> # Commencez par créer un objet Evenement.
    >>> evenement = Evenement()
    >>> evenement.nom="Rendez-vous chez le dentiste"
    >>> evenement.description = "Pas très agréable :("
    >>> evenement.date = datetime.now() + timedelta(days = 5)
    >>> evenement.lieu = "Chez le dentiste"
    >>> # Sauvegardez l'objet Evenement.
    >>> evenement.save()
    >>> # Vérifiez que l'objet a bien été créé.
    >>> rendezvous = Evenement.objects.get(lieu="Chez le dentiste")
    >>> print rendezvous
    Evenement object
    >>> # Vérifiez l'un des champs de l'objet.
    >>> rendezvous.nom
    u'Rendez-vous chez le dentiste'
    """
```

Premier test

Maintenant que votre test est écrit, il faut bien entendu l'exécuter. Django vous offre un système très simple pour lancer vos tests. Il suffit de lancer la commande suivante :

```
python manage.py test personal_calendar
Creating test database for alias 'default'.....
----------------------------------------------------------------------
Ran 2 tests in 0.031s

OK
Destroying test database for alias 'default'...
```

Si le résultat est différent, Django devrait vous indiquer votre erreur. Par exemple, si vous changez la dernière ligne de votre docstring en indiquant `u'Rendez-vous chez le medecin'` au lieu de `u'Rendez-vous chez le dentiste'`, voici quelle sera la réponse de Django :

```
Creating test database for alias 'default'...
.F
======================================================================= =======
FAIL: Evenement (personal_calendar.models)
Doctest: personal_calendar.models.Evenement
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/user/env_projet/lib/python2.7/site-packages/dja go/test/
_doctest.py", line 2180, in runTest
   raise self.failureException(self.format_failure(new.getvalue()))
AssertionError: Failed doctest test for personal_calendar.models.Evenement
  File "/home/user/env_projet/agenda/personal_calendar/models.py", line 19, in
Evenement

----------------------------------------------------------------------
File "/home/user/env_projet/agenda/personal_calendar/models.py", line 58, in
personal_calendar.models.Evenement
Failed example:
    rendezvous.nom
Expected:
    u'Rendez-vous chez le medecin'
Got:
    u'Rendez-vous chez le dentiste'


----------------------------------------------------------------------
Ran 2 tests in 0.046s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Comme vous le voyez, Django fournit une trace complète de l'erreur, vous permettant de corriger immédiatement.

Base de données de test

Vous avez peut-être été intrigué, lors du lancement de vos tests, par la première ligne du retour de Django :

```
Creating test database for alias 'default'...
```

En effet, à chaque fois que vous lancez vos tests, Django crée pour vous une base de données temporaire le temps du test. Plus loin, vous verrez comment manipuler cette base de données.

À titre d'exercice, vous pouvez écrire les tests des modèles de l'application tracker et ceux du modèle `Evenement_Participant`. C'est sans doute un peu compliqué, car cela vous demande de créer un objet `User`, puis un objet `Evenement` et enfin un objet `Evenement_Participant`, ce qui risque d'être un peu long pour une docstring. Malgré tout, n'hésitez pas à le faire, car c'est très formateur.

Quand vous créez vos tests de cette façon, n'hésitez pas à faire l'aller-retour entre le shell de Django et la docstring de vos modèles. Au fur et à mesure que vous écrirez vos modèles et les tests qui les accompagnent, les tests précédemment écrits seront eux aussi lancés. Les tests vont ainsi vous aider à valider que vos modifications n'entraînent pas de régressions.

Les tests unitaires

Pour rédiger des tests plus complexes, Django nous offre un autre outil avec les tests unitaires (*unittests* en anglais). Avez-vous remarqué que dans chaque application créée via la commande `python manage.py startapp`, Django avait ajouté pour vous un fichier test ? Rendez-vous dans le fichier `test` de l'application `personal_calendar` et regardez ce qu'il contient.

```
"""
This file demonstrates writing tests using the unittest module. These will pass
when you run "manage.py test".

Replace this with more appropriate tests for your application.
"""

from django.test import TestCase


class SimpleTest(TestCase):
    def test_basic_addition(self):
        """
        Tests that 1 + 1 always equals 2.
        """
        self.assertEqual(1 + 1, 2)
```

Tout d'abord, remarquez l'usage d'une docstring en début de fichier. Elle vous indique ce que fait ce module et comment l'utiliser.

Le module test de votre application ne contient pour l'heure qu'un import :

```
from django.test import TestCase
```

une classe :

```
class SimpleTest(TestCase):
```

et une fonction :

```
def test_basic_addition(self):
```

Pour écrire vos propre tests dans ce fichier, il vous suffit de créer une classe héritant de `TestCase` importée depuis le module `django.test` et d'y définir des fonctions dont le nom commence par le mot-clé `test`. Lorsque les tests sont lancés, Django vérifie que les fonctions ont bien le comportement attendu. Dans l'exemple fourni, la méthode `assertEqual` est utilisée : elle renvoie `True` si les deux expressions données en argument sont équivalentes et renvoie `False` dans le cas contraire.

Pour exemple, modifiez la fonction `test_basic_addition` : au lieu de `self.assertEqual(1 + 1, 2)`, écrivez `self.assertEqual(1 + 1, 3)` et relancez vos tests :

```
python manage.py test personal_calendar
Creating test database for alias 'default'...
F.
======================================================================
FAIL: test_basic_addition (personal_calendar.tests.SimpleTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/user/env_projet/agenda/personal_calendar/tests.py",
  line 16, in test_basic_addition
    self.assertEqual(1 + 1, 3)
AssertionError: 2 != 3

----------------------------------------------------------------------
Ran 2 tests in 0.031s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Il est temps maintenant d'écrire de vrais tests.

Test unitaires des modèles

La première chose que vous allez tester, c'est le comportement de vos modèles. Si les tests dans les docstrings vous permettaient de vérifier succinctement vos modèles et d'en faire une documentation par l'exemple, ici vous pouvez écrire des tests bien plus complets. Le tout est de rester méthodique. Créez une classe par modèle présent dans le fichier `models.py`, chacune héritant de `TestCase`. Par exemple, pour la classe `Evenement_Participant`, vous pouvez écrire :

```
class TestEvenement_Participant(TestCase)
```

Bien entendu, en début de fichier, il vous faut importer votre modèle :

```
from personal_calendar import Evenement_Participant
```

Le modèle `Evenement_Participant` étant la jonction entre un utilisateur et un événement, il va aussi falloir importer les modèles `User` et `Evenement`, ainsi que `datetime.datetime` et `datetime.timedelta` dont vous allez avoir besoin. Votre fichier devrait ressembler à ceci :

```
#-*-coding:utf-8 -*-
fromdjango.testimportTestCase
from personal_calendar.models import Evenement_Participant, Evenement
from django.contrib.auth.models import User
from datetime import datetime
from datetime import timedelta

class TestEvenement_Participant(TestCase):
```

Une fois que tout est prêt, écrivez votre première fonction de test :

```
def test_create_user(self):
    """
    Nous avons besoin d'un utilisateur en tant que participant à un
    événement, essayons d'en créer un.
    """
    participant1 = User(first_name ="John", last_name = "Doe")
    participant1.save()
    # Testons ensuite que cet élément a bien été enregistré.
    participant = User.objects.get(first_name = "John")
    self.assertEqual(participant.last_name, "Doe")
```

C'est suffisant pour l'instant. Vous venez d'écrire un test qui vérifie l'enregistrement d'un objet `User`. Lancez votre test et vérifiez qu'il n'échoue pas.

Maintenant, testez l'enregistrement d'un objet `Evenement` :

```
    def test_create_event(self):
        evenement = Evenement(nom='Un nouvel événement',
                              description="""Ce nouvel événement est
créé pour montrer le fonctionnement des test unitaires.
Comme nous ne sommes plus dans une docstring, il est bien plus facile
        de faire des chaînes de caractères qui s'étendent sur plusieurs
        lignes. On peut également utiliser un format plus compact."""
    # Ici, on se ressert de l'objet Participant créé dans la
    # première fonction.
                                date=datetime.now(),
                                lieu="Un lieu quelconque")
        evenement.save()
        evenement = Evenement.objects.get(nom="Un nouvel événement")
        self.assertEqual(evenement.lieu, u'Un lieu quelconque')
```

Une fois encore, vous pouvez lancer vos tests et vérifier que tout se passe bien.

Il ne nous reste plus qu'à tester notre dernier modèle : `Evenement_Participant`.

```
    def test_create_event_participant(self):
        self.test_create_user()
        self.test_create_event()
        participant = User.objects.get(first_name="John")
        evenement = Evenement.objects.get(nom='Un nouvel événement')
        evnt_part = Evenement_Participant(evenement=evenement,
                                          participant=participant,
                                          status=1)
        evnt_part.save()
        evnt = Evenement_Participant.objects.get(evenement=evenement,
                                                 participant=participant
                                                 )
        self.assertEqual(evnt.status, 1)
```

Lancez vos tests et constatez que tout se passe comme prévu. Vous venez non seulement de vérifier le comportement correct de vos modèles, mais aussi de vous offrir une sorte de « pense-bête » sur la façon de les employer. Un nouveau développeur qui lira votre code comprendra immédiatement comment utiliser la table de jointure `Evenement_Participant` par exemple.

Ce que nous venons de voir à propos des tests unitaires ne couvre que la surface des choses. En effet, les tests écrits ici ne sont pas complets.

Rev 3

Ajout des tests unitaires et des docstrings.

Conclusion

Au cours de ce chapitre vous avez pu en apprendre plus sur la manière dont fonctionne Django et comment préparer un projet de la meilleure façon qui soit. Vous avez également appris à écrire et utiliser des tests unitaires et des docstrings. Même si cela peut vous sembler fastidieux, il est crucial que vous fassiez l'effort d'écrire ces tests et ces docstrings. Garants de la qualité et de la pérennité de votre code, ils vous permettront de vous améliorer en permanence en tant que développeur professionnel.

4

Gestion de l'authentification

Dans le développement web, l'authentification des utilisateurs est un problème récurrent. Comme Django est un framework pragmatique, il propose un système simple, performant et robuste pour authentifier vos utilisateurs. Dans de nombreux cas, il vous faudra créer, gérer et authentifier les utilisateurs de votre site, à la fois pour personnaliser votre contenu et pour restreindre l'accès à certaines données.

Authentifier un utilisateur sur le site

Avant d'étudier les aspects plus avancés de la gestion des utilisateurs, il faut déjà que le site que vous concevez soit en mesure de reconnaître un visiteur. La technique la plus couramment employée est de proposer un formulaire avec les champs nom d'utilisateur et mot de passe. Django propose une mise en place de ce type de page en seulement deux étapes.

1. Ajouter les URL `login` et `logout`.
2. Créer les templates `login` et `logout`.

Ajouter les URL login et logout

Dans l'application `auth` des contribs, vous trouverez un module `views` qui contient déjà les fonctions `login` et `logout`. Il vous suffit de les appeler dans vos URL pour les rendre immédiatement disponibles. Elles gèrent pour vous tout ce dont vous avez

besoin pour identifier un utilisateur. Voici comment vont se présenter vos URL (fichier `urls.py` dans le dossier `agenda`) :

```
from django.conf.urls.defaults import patterns, include, url
from django.contrib.auth.views import login, logout
# Uncomment the next two lines to enable the admin:
# from django.contrib import admin
# admin.autodiscover()

urlpatterns = patterns('',
   # Examples:
    # url(r'^$', 'agenda.views.home', name='home'),
    # url(r'^agenda/', include('agenda.foo.urls')),

    # Uncomment the admin/doc line below to enable admin documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # url(r'^admin/', include(admin.site.urls)),
    url(r'^accounts/login/$', login),
    url(r'^accounts/logout/$', logout, {'next_page':'/accounts/login/'}),
)
```

Remarque

L'URL `url(r'^accounts/logout'/$, logout, {'next_page':'/accounts/login/'})` est un peu différente de celles que vous avez pu voir auparavant. Elle contient un dictionnaire `{'next_page':'/accounts/login/'}` qui va être utilisé par la vue `login` pour connaître l'URL vers laquelle il faut rediriger un utilisateur après qu'il s'est déconnecté.

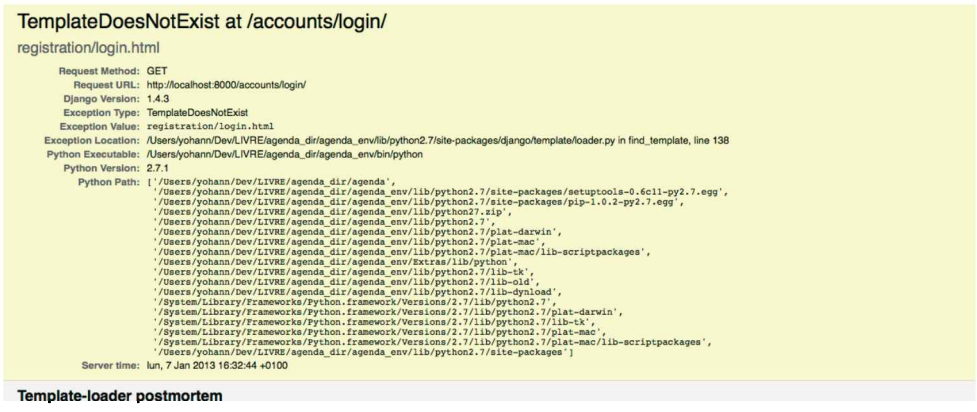


Figure 4–1 Le template n'existe pas.

Si vous testez ces URL en lançant le serveur de développement de Django et en accédant à `localhost:8000/accounts/login`, vous verrez s'afficher l'écran de la page précédente (voir figure 4-1).

Créer le template login

Comme vous le signale Django, il vous manque le template `registration/login.html`. Créez le dossier `registration` dans le dossier `templates` à la racine de votre projet. Vous avez déjà indiqué à Django où se trouvent vos templates dans `settings.py` :

```
TEMPLATE_DIRS = (
    project_path + 'templates'
)
```

Dans le dossier `registration`, il vous faut créer un fichier `login.html`, contenant le formulaire HTML de login qui a déjà été préparé par la vue `login` de `django.contrib.auth.views`. Il vous suffit de l'appeler de la façon suivante :

```
{{form.as_p}}
```

Ici, il est demandé à Django d'afficher le formulaire sous forme de paragraphe (`as_p`), mais vous pouvez aussi l'obtenir sous forme de tableau (`as_table`) ou de liste (`as_ul`). Plus loin, nous verrons également comment prendre totalement le contrôle de l'affichage (voir chapitre 13 « Dialogue avec l'utilisateur : les formulaires »). Pour le moment, affichez dans votre navigateur le formulaire que vous venez de créer.

Problème : ce formulaire est inutile pour le moment, car il n'y a pas de bouton pour le valider. Pour le rendre pleinement fonctionnel, vous avez besoin de deux choses :

- la balise HTML `form` ;
- le bouton HTML `submit`.

Django n'a pas créé ces deux éléments, car son principe est de vous simplifier la vie et non de vous mettre des bâtons dans les roues. En effet, s'il vous proposait la balise `form` et le bouton `submit`, vous auriez toutes les peines du monde à modifier le comportement d'un formulaire, par exemple pour le faire valider sur une nouvelle URL ou pour utiliser un lien pour la validation au lieu d'un bouton `submit`.

Voici donc le formulaire avec les deux éléments qui vous manquaient :

```
<form action = "" method = "POST">
{{form.as_p}}
<input type ="submit" value ="Login" />
</form>
```

Affichez de nouveau la page `http://localhost:8000/registration/login/` et tentez de remplir le formulaire et de le valider. Visiblement, quelque chose ne va pas ! Django se plaint de ne pas avoir le template tag `{% csrf_token %}`.

SÉCURITÉ csrf_token

CSRF signifie *Cross Site Request Forgery*. Il s'agit d'une attaque qui consiste à se servir, à son insu, de la session de l'utilisateur. Imaginez que vous êtes connecté à l'interface d'administration de Django. Un webmestre malveillant vous invite à vous rendre sur l'une de ses pages. Lorsque vous vous y connectez, il se sert de votre session pour se connecter lui aussi à l'administration de Django sous votre compte.
Pour éviter ce type d'attaque, Django a mis au point une technique très simple. Elle consiste à ajouter une chaîne de caractères pour chaque formulaire de type POST, qui est vérifiée et ne peut être utilisée qu'une seule fois. C'est le rôle du template tag `{% csrf_token %}`.

Ajoutez donc tout simplement ce template tag à votre template :

```
<form action = "" method = "POST">
{% csrf_token %}
{{form.as_p}}
<input type ="submit" value ="Login" />
</form>
```

et testez de nouveau votre formulaire.

C'est beaucoup mieux. Toutefois, Django ne trouve pas l'URL `accounts/profile`, ce qui est parfaitement normal puisque vous ne l'avez pas encore écrite. Pour le moment, revenez à votre formulaire ; tentez d'y inscrire des données erronées et voyez comment il se comporte.

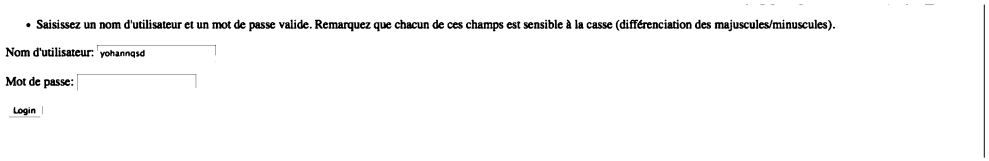

• Saisissez un nom d'utilisateur et un mot de passe valide. Remarquez que chacun de ces champs est sensible à la casse (différenciation des majuscules/minuscules).
Nom d'utilisateur: yohannqsd
Mot de passe:
Login

Figure 4–2 La validation de formulaire

Comme vous pouvez le remarquer, Django gère déjà pour vous toute la validation du formulaire et le traitement des erreurs. En tout et pour tout, vous avez eu à écrire moins d'une dizaine de lignes de code pour mettre en place un formulaire de connexion.

Personnaliser le contenu

Vous allez maintenant utiliser les informations dont vous disposez pour personnaliser le contenu affiché à l'utilisateur.

Une page pour chaque utilisateur

Après avoir identifié votre utilisateur, Django le redirige vers l'URL `/accounts/profile/` qui n'existe pas. Ce comportement est bien entendu totalement configurable :

- soit globalement pour tout le site, en ajoutant la ligne suivante au `settings.py` :

```
LOGIN_REDIRECT_URL = "l'URL vers laquelle un utilisateur est redirigé après s'être identifié"
```

- soit localement, en ajoutant un champ caché à votre formulaire :

```
<input type="hidden" name="next" value="l'URL de redirection" />
```

Dans le cas présent, l'URL `/accounts/profile/` est tout indiquée ; il n'y a donc rien à modifier. Il vous reste donc trois choses à faire :

- ajouter `/accounts/profile/` aux URL ;
- créer une vue qui va charger les données de l'utilisateur et l'afficher dans un template ;
- créer le template qui va recevoir les données de l'utilisateur et les afficher.

En somme, il s'agit ici d'un workflow MVC (voir chapitre 2).

Ajoutez donc à vos URL :

```
url(r'^accounts/profile/$', profile),
```

Au début du fichier `urls.py`, importez la fonction `profile` :

```
from personal_calendar.views import profile
```

Et créez la vue `profile` dans le fichier `agenda/personal_calendar/views.py` :

```
from django.shortcuts import render
def profile(request):
    return render(request, 'registration/profile.html', {})
```

Enfin, vous pouvez créer votre template `registration/profile` :

```
<h1> Bienvenue sur votre page personnelle </h1>
<a href = "/accounts/logout/">Se déconnecter</a>
```

Vous pouvez tester cette page, vous connecter et vous déconnecter. Vous constatez que le mécanisme est parfaitement fonctionnel, tout ceci avec relativement peu de code.

Rev 4

Un premier système d'authentification fonctionnel.

Pourtant, plusieurs problèmes se posent.

- La fonction `profile` se trouve dans l'application `personal_calendar` alors qu'elle devrait être disponible globalement sur le site. Ainsi, si un développeur externe veut utiliser l'application, la fonction `profile` ne lui sera d'aucune utilité. Pire, si vous souhaitez désactiver cette application, c'est tout le processus d'authentification qui sera cassé.
- La fonction `profile` n'est qu'un bouchon entre l'URL et le template ; elle ne fait rien d'utile. Il serait plus intelligent de la supprimer totalement.
- Le template `profile.html` ne donne aucune information à l'utilisateur. En l'état actuel, il lui permet seulement de se déconnecter.

Voyons comment résoudre les deux premiers problèmes en écrivant moins de code.

Une introduction aux vues génériques : TemplateView

Lier une URL à un template sans passer par une vue est une action assez fréquente dans le développement web. Là encore, Django vous simplifie la vie : un module complet gère les cas des vues génériques. Ces dernières répondent à un besoin simple, récurrent et fastidieux. On peut citer l'affichage « direct » d'un template, l'affichage d'une liste d'objets, des détails d'un objet unique, etc. Pour chacun de ces besoins simples, une vue générique est disponible. Vous trouverez l'ensemble des vues génériques dans le module `django.views.generic`.

Pour afficher le template `profile`, vous allez utiliser la classe `TemplateView`. Il suffit de l'importer dans vos URL :

```
from django.views.generic import TemplateView
```

Ensuite, remplacez :

```
url(r'^accounts/profile/$', profile),
```

par :

```
    url(r'^accounts/profile/$',
        TemplateView.as_view(template_name="registration/profile.html")),
```

Après avoir testé le bon fonctionnement de cette nouvelle URL, vous pouvez supprimer l'`import` de la fonction `profile` :

```
from personal_calendar.views import profile
```

Bien entendu, vous pouvez aussi effacer la fonction `profile` de votre fichier `personal_calendar/views.py`.

Rev 5

Un système d'authentification moins invasif.

Afficher les informations de l'utilisateur connecté

Ce point peut être résolu tout aussi simplement. En effet, la fonction `TemplateView.as_view()` que vous venez d'utiliser rend disponible au template qu'elle charge le `Context` de la requête. Ce dernier est un dictionnaire contenant, entre autre, l'utilisateur connecté. Il suffit donc de modifier très légèrement le template `profile` pour personnaliser un peu la page :

```
<h1> Bienvenue sur votre page personnelle {{user}}</h1>
<a href = "/accounts/logout/">Se déconnecter</a>
```

C'est très simple, mais vous mettez pour l'instant tout en place pour que la suite du développement de votre application d'agenda partagé se passe le mieux possible.

Autoriser ou interdire certaines parties du site

L'intérêt principal d'une gestion de l'authentification réside dans la possibilité d'interdire une partie du site aux personnes non authentifiées. Avec Django, les autorisations peuvent être gérés au niveau des URL, des vues ou des templates. Voyons tout d'abord comment protéger une URL.

Dans le cas présent, l'URL `/account/profile/` n'est pas protégée. Ainsi, n'importe quel utilisateur non identifié peut donc s'y rendre. Pour le moment, ce n'est pas un problème car aucune information sensible n'est disponible, mais cela peut le devenir

au cours des évolutions de l'application. Le plus simple reste donc de protéger dès maintenant cette URL en utilisant un simple décorateur proposé par Django : `login_required`. Sa mise en place est enfantine.

Importez d'abord le décorateur dans vos URL :

```
from django.contrib.auth.decorators import login_required
```

Puis « entourez » l'URL que vous souhaitez protéger dans la fonction `login_required` :

```
    url(r'^accounts/profile/$',
        login_required(
            TemplateView.as_view(template_name="registration/profile.html"))),
)
```

> **DÉFINITION Les décorateurs Python**
>
> Schématiquement, les décorateurs sont des fonctions qui prennent en argument une fonction et renvoient une fonction.

Maintenant que les utilisateurs peuvent s'authentifier sur le site, il faut également implémenter une méthode pour qu'ils créent un compte. Elles sont nombreuses pour créer les utilisateurs, via l'administration de Django ou la ligne de commande avec la commande `python manage.py shell`. D'autres, plus complexes, peuvent aussi être utilisées, qui comprennent l'envoi d'un e-mail avec un lien de validation. Pour l'heure, nous allons simplement bâtir un formulaire permettant de créer un utilisateur directement utilisable.

Découplage de l'application

Pour que votre projet reste découplable, il faut séparer la création des utilisateurs de la logique de votre application `personal_calendar`. Vous allez donc réaliser une nouvelle application qui se chargera de gérer la création des utilisateurs. Ainsi, vous pourrez également l'employer sur un autre projet (le tracker, par exemple). Vous garderez aussi la possibilité de choisir un autre système de création de compte pour votre projet `agenda` sans devoir changer une ligne de l'application `personal_calendar` (`django-registration` par exemple, voir https://bitbucket.org/ubernostrum/django-registration/).

Commencez par créer cette nouvelle application :

```
python manage.py startapp usermanagement
```

Ensuite, ajoutez-la au `settings.py` de votre projet.

Pour utiliser les vues – les fonctions que vous écrirez dans le fichier `views.py` –, vous pourriez ajouter de nouvelles URL au fichier `agenda/urls.py`. Pourtant, cela aurait le désavantage de rendre délicat le découplage de votre application. Django propose une méthode beaucoup plus souple avec la fonction `include` qui enrichit les URL de votre projet avec les fichiers d'URL de vos applications. Voici comment elle est utilisée :

```
url(<expression régulière>, include(<le chemin des URL à inclure>))
```

Prenons, par exemple, le code suivant :

```
url(r'^foo/', include('agenda.foo.urls')),
```

Lors du processus de résolution des URL, lorsque Django rencontrera une URL commençant par `/foo`, il chargera le fichier `agenda/foo/urls` et continuera la résolution à partir de ce fichier. Pour mieux comprendre ce processus, prenons un exemple concret.

Créez le fichier `agenda/usermanagement/urls.py` :

```
from django.conf.urls.defaults import patterns, include, url
urlpatterns = patterns('',
)
```

Dans le fichier `agenda/urls.py`, modifiez :

```
from django.conf.urls.defaults import patterns, url
```

en

```
from django.conf.urls.defaults import patterns, include, url
```

puis ajoutez une nouvelle URL :

```
url(r'^user/', include('usermanagement.urls')),
```

Désormais, toutes les URL commençant par `user` seront redirigées vers le fichier `agenda/usermanagement/urls.py`. Pour le vérifier, créez l'URL suivante dans le fichier `agenda/usermanagement/urls.py` :

```
from django.conf.urls.defaults import patterns, include, url
from django.views.generic import TemplateView
urlpatterns = patterns('',
url(r'^succes/$',TemplateView.as_view(template_name="user/succes.html"))
)
```

Cette URL reprend la vue générique `TemplateView` qui redirige l'utilisateur vers un template. Elle servira à afficher un message de succès à l'utilisateur lors de la création de compte. Le template `user/succes.html` peut ressembler à ceci :

```
<h1>Votre compte a bien été créé.</h1>
<p>Vous pouvez désormais vous <a href="/accounts/login/">connecter</a></p>
```

Rendez-vous ensuite à l'adresse `http://localhost:8000/user/succes/` pour constater que Django a analysé l'URL `/user/succes/` de la façon suivante.

- L'URL débute par `/user/`. Il faut donc regarder dans le fichier `usermanagement.urls.py` pour continuer la résolution.
- Le reste de l'URL, `/succes/`, coïncide avec la ligne :

```
url(r'^succes$',TemplateView.as_view(template_name="user/succes.html"))
```

Le template `user/succes.html` est alors chargé.

Rev 8

La page /user/succes/.

Pour réaliser la vue qui va présenter le formulaire d'inscription, les choses vont être un peu différentes. Vous allez devoir :

- instancier un formulaire ;
- valider les données du formulaire ;
- enregistrer les données du formulaire ;
- présenter les erreurs du formulaire si besoin.

Instancier un formulaire

Django vous fournit un formulaire pour créer un nouvel utilisateur ; vous n'avez donc pas à le faire vous-même. Le formulaire `UserCreationForm` réside dans le module `django.contrib.auth.forms`. Dans le module `views` de l'application `usermanagement`, importez donc cette classe :

```
from django.contrib.auth.forms import UserCreationForm
```

Il faut ensuite créer une fonction qui va présenter ce formulaire :

```
from django.contrib.auth.forms import UserCreationForm
from django.shortcuts import render
from django.http import HttpResponseRedirect

def create_account(request):
    form = UserCreationForm()
    return render(request,'user/create.html',{'form':form})
```

Il vous faut bien entendu créer le template `create.html` dans le dossier `/templates/user/` :

```
<form action ="" method ="POST" >
{%csrf_token%}
{{form.as_p}}
<input type="submit" value="Valider">
</form>
```

Remarquez que ce formulaire est similaire en tout point à celui réalisé pour l'identification de l'utilisateur. Il ne reste plus qu'à brancher cette nouvelle fonction dans le fichier `urls.py` de l'application `usermanagement` :

```
from views import create_account
urlpatterns = patterns('',
<...>
    url(r'^create_account/$', create_account),
```

Testez ensuite votre formulaire à l'adresse `http://localhost:8000/user/create_account`.

Valider les données du formulaire

Comme vous le constatez, le formulaire s'affiche correctement. Seulement, pour le moment, vous ne pouvez rien en faire. Lorsque vous le soumettez, que vos données soient erronées ou valides, la page est rechargée et un nouveau formulaire vide est affiché. C'est d'ailleurs le comportement normal de votre fonction, qui retourne un formulaire vide quelle que soit la requête :

```
form = UserCreationForm()
```

Dans le template que vous venez de créer, vous indiquez que le formulaire doit être soumis avec la méthode `POST` :

```
<form action ="" method ="POST" >
```

Django peut entreprendre une action différente si la requête utilise le verbe `POST` – nous préciserons plus loin l'usage des verbes du Web. Pour le vérifier, demandez à Django d'afficher un message dans le cas où la requête est de type `POST` :

```
def create_account(request):
    if request.method == "POST":
        print "c'est bien du POST"
    form = UserCreationForm()
    return render(request,'user/create.html',{'form':form})
```

La condition en exergue va afficher dans votre terminal le texte `c'est bien du POST` à chaque validation du formulaire.

Il faut maintenant récupérer les données de ce formulaire grâce à la variable `request.POST` :

```
def create_account(request):
    if request.method == "POST":
        print request.POST
...
```

La classe `UserCreationForm`, comme tous les formulaires Django, propose la méthode `is_valid()`, qui sert à valider chaque champ et le formulaire dans son ensemble, afin de vérifier que les données de l'utilisateur correspondent bien à ce que vous attendez. Pour instancier un formulaire avec les données fournies, il suffit d'écrire :

```
form = UserCreationForm(request.POST)
```

Vous pouvez alors valider votre formulaire de la façon suivante :

```
if form.is_valid():
    print "valide"
```

Enregistrer les données du formulaire

L'enregistrement du formulaire fait lui aussi appel à une méthode de la classe `UserCreationForm` ; il s'agit de `save()` :

```
if form.is_valid():
    form.save()
```

Une fois que le formulaire est sauvegardé – Django se charge pour vous de l'enregistrement du nouveau compte en base de données –, vous pouvez rediriger votre nouvel utilisateur vers l'URL `/user/succes/` que vous avez créée auparavant.

La fonction create_account finalisée

Voici comment se présentera votre fonction `create_account` :

```
from django.contrib.auth.forms import UserCreationForm
from django.shortcuts import render
from django.http import HttpResponseRedirect

def create_account(request):
    if request.method == "POST":
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/user/succes/')
    else:
        form = UserCreationForm()
    return render(request,'user/create.html',{'form':form})
```

Rev 9

La création de compte.

Les tests unitaires de l'authentification

Vous avez désormais un système d'authentification pleinement fonctionnel :

- création de nouveaux utilisateurs ;
- connexion des utilisateurs ;
- déconnexion des utilisateurs ;
- affichage des informations d'un utilisateur.

Toutefois, vous n'avez réalisé aucun test. Or, pour que la technique des tests fonctionnels soit efficiente, il faut que chaque partie de votre code soit vérifiée. Les techniques que vous avez vues précédemment ne permettent de tester que les modèles. Pour les vues, il vous faut un système qui se comporte comme un utilisateur, c'est-à-dire qui demande d'accéder à une URL, qui s'enregistre, crée un compte et fait des assertions sur la réponse. Django propose un tel système via la classe `Client`.

La classe Client

La classe `Client` de Django se présente de manière très simple et sert à tester toute la logique de votre site. Elle n'a pas besoin d'un serveur de développement et s'utilise en association avec les tests unitaires classiques. Pour vérifier la logique d'authentification, nous vous proposons d'écrire les tests suivants.

Une première série de tests validera simplement l'existence des URL principales.

- `/accounts/login/` : ce test devra renvoyer le code `200` (pour les *status codes*, voir le chapitre 4 « Django et le protocole web »).
- `/accounts/logout/` ce test renverra le code `302` (redirection).
- `/accounts/profile/` : ce test renverra également le code `302`.

La seconde série de tests validera le comportement des formulaires :

- `/accounts/login/` avec des données invalides renverra le code `200` et le template `login.html`.
- `/accounts/login/` avec des données valides renverra le code `200` et le template `/accounts/profile`.
- `/user/create_account` avec des données invalides renverra le code `200` et le template `create_account.html`.
- `/user/create_account` avec des données valides renverra le code `200` et le template `succes.html`.

Enfin, on pourra réaliser un test plus fonctionnel :

- création d'un nouvel utilisateur via le formulaire ;
- test de la présence du compte en base de données ;
- connexion de l'utilisateur via le formulaire `login` ;
- tentative d'accès à la page `/accounts/profile/`.

Avec ces trois batteries de tests, vous aurez une couverture complète de votre système d'authentification. Comme les précédents, nous vous conseillons de les écrire dans le module `test` de l'application `usermanagement`.

Test des URL

Pour tester vos URL, le plus simple est de vérifier les *status* HTTP que vous renvoie Django. Pour cela, il faut utiliser la classe `Client` de `django.test` et créer la classe qui va tester les status, puis instancier un objet `Client`. Comme ce dernier va vous servir tout au long des tests sur les status, il est plus simple de définir dès maintenant la fonction `setUp`, qui est spéciale, un peu comme `__init__` ; elle va être lancée avant les tests. Vous pouvez y définir un environnement de test qui sera utilisé pour tous les tests de la classe :

```
from django.test import TestCase
from django.test import Client

class StatusTest(TestCase):
    def setUp(self):
        self.client = Client()
```

Le premier test consiste à vérifier l'existence de l'URL `/accounts/login/`. Pour cela, il suffit d'utiliser la méthode `get` de `Client` :

```
response = self.client.get('/accounts/login/')
```

qui vous renvoie un objet `HttpResponse`. Vous avez donc à votre disposition l'attribut `status_code` de cette réponse et vous pouvez faire une assertion du type :

```
self.assertEqual(response.status_code, 200)
```

Votre fichier `test.py` devrait donc ressembler à ceci :

```
from django.test import TestCase
from django.test import Client

class StatusTest(TestCase):
    def setUp(self):
        self.client = Client()

    def test_login(self):
        response = self.client.get('/accounts/login/')
        self.assertEqual(response.status_code, 200)
```

Bien entendu, vous pouvez écrire une fonction différente pour chacune des URL que vous souhaitez tester. Toutefois, cela risque de devenir un peu répétitif et de produire un code assez laid. Nous vous proposons donc de simplifier les choses de la façon suivante :

```
def test_public(self):
    urls = [{'url':'/accounts/login/','status':200},
            {'url':'/accounts/logout/','status':302},
            {'url':'/accounts/profile/','status':302}
           ]
    for elem in urls:
        response = self.client.get(elem['url'])
        self.assertEqual(response.status_code, elem['status'])
```

Grâce à ce dictionnaire, un seul test suffira pour vérifier le comportement de vos URL. Pourtant, `/accounts/logout/` et `/accounts/profile/` ne sont pas totalement

testées : vous avez vérifié qu'elles occasionnaient une redirection, mais pas qu'elles vous redirigeaient vers la bonne URL.

Pour le moment, la méthode `get` de `Client` ne suit pas les redirections. Pour modifier ce comportement, il faut utiliser le paramètre `follow=True` et tester les templates des réponses :

```
    def test_public(self):
        urls = [{'url':'/accounts/login/',
                 'template':'registration/login.html',
                 'status':200
                },
                {'url':'/accounts/logout/',
                 'template':'registration/login.html',
                 'status':302},
                {'url':'/accounts/profile/',
                 'template':'registration/login.html',
                 'status':302}
                    ]
        for elem in urls:
            response = self.client.get(elem['url'])
            self.assertEqual(response.status_code,elem['status'])
            response = self.client.get(elem['url'],follow=True)
            self.assertEqual(response.template.name, elem['template'])
```

Les tests des formulaires

Les formulaires de votre application emploient le verbe `POST`. Pour les tester, vous devez donc utiliser la méthode `post` de la classe `Client`, qui prend les même arguments que la méthode `get`. Pour y inclure des données de formulaire, il faut lui fournir en paramètre un dictionnaire ayant pour clés les noms des champs et pour valeurs les données à valider. Pour le formulaire de création de compte, cela donne :

```
        response = self.client.post('/user/create_account/',{
                'username':'john',
                'password1':'trytoguess',
                'password2':'trytoguess'
                },follow=True
```

En toute logique, une fois le formulaire validé, le dernier template affiché devrait être `user/succes.html` :

```
    def test_create_user(self):
         response = self.client.post('/user/create_account/',{
                'username':'john',
```

```
                'password1':'trytoguess',
                'password2':'trytoguess'
                },follow=True
                                    )
        self.assertEqual(response.template.name,'user/succes.html')
```

Maintenant, il reste à s'assurer que l'utilisateur en question a bien été créé. Pour cela, il faut revenir aux tests usuels :

```
user = User.objects.get(username="john")
self.assertEqual(user.username,"john")
```

Il reste encore un dernier test de formulaire à effectuer, celui de `login`. La logique reste la même que pour le formulaire de création d'utilisateur :

```
def test_login(self):
    self.test_create_user()
    response = self.client.post('/accounts/login/',{
                'username':'john',
                'password':'trytoguess'
                },follow=True)
     self.assertEqual(response.template.name,
                      'registration/profile.html'
                     )
```

Voici l'intégralité du test :

```
from django.test import TestCase
from django.test import Client
from django.contrib.auth.models import User
class StatusTest(TestCase):
    def setUp(self):
        self.client = Client()

    def test_public(self):
        urls = [{'url':'/accounts/login/',
                 'template':'registration/login.html',
                 'status':200
                },
                {'url':'/accounts/logout/',
                 'template':'registration/login.html',
                 'status':302},
                {'url':'/accounts/profile/',
                 'template':'registration/login.html',
                 'status':302}
                    ]
```

```
        for elem in urls:
            response = self.client.get(elem['url'])
            self.assertEqual(response.status_code,elem['status'])
            response = self.client.get(elem['url'],follow=True)
            self.assertEqual(response.template.name, elem['template'])

    def test_create_user(self):
         response = self.client.post('/user/create_account/',{
                 'username':'john',
                 'password1':'trytoguess',
                 'password2':'trytoguess'
                 },follow=True)
         self.assertEqual(response.template.name,'user/succes.html')
         user = User.objects.get(username="john")
         self.assertEqual(user.username,"john")

    def test_login(self):
        self.test_create_user()
        response = self.client.post('/accounts/login/',{
                'username':'john',
                'password':'trytoguess'
                },follow=True)
        self.assertEqual(response.template.name,'registration/profile.html')
```

Rev 10

Ajout des tests unitaires.

Les tests d'échec

Tester que les choses fonctionnent quand cela doit être le cas, c'est une bonne chose. Mais il ne faut pas oublier de toujours tester les erreurs. Que se passerait-il, par exemple, si le formulaire était invalide ?

À titre d'exercice, nous vous laissons réaliser vous-même ces tests. Il vous faudra les écrire de la même façon que ceux que nous avons construits ensemble. Idéalement, il faut vérifier :

- que deux mots de passe différents renvoient le formulaire et ne créent pas d'utilisateur.
- qu'une tentative de connexion avec un mot de passe erroné se solde par un échec.

Derniers ajustements

Votre application `usermanagement` est terminée, mais il reste quelques détails à peaufiner.

Modifier les URL

Votre application `usermanagement` est désormais terminée. Pourtant, si vous souhaitez l'utiliser dans un autre projet, vous allez être confronté à un petit problème : les URL `login` et `logout` sont définies à la racine de votre projet ; il faudra donc les ajouter à vos URL principales avant de pouvoir les utiliser. Pour vous éviter ce travail fastidieux, il est plus simple de déplacer ces URL vers l'application `usermanagement`. Cela se fait très simplement et c'est également une très bonne introduction à la méthode du TDD (voir chapitre 2).

Avant de modifier vos URL, changez vos tests. Par exemple :

```
    def test_public(self):
        urls = [{'url':'/user/login/',
                 'template':'registration/login.html',
                 'status':200
                },
                {'url':'/user/logout/',
                 'template':'registration/login.html',
                 'status':302},
                {'url':'/user/profile/',
                 'template':'registration/login.html',
                 'status':302}
                    ]
        for elem in urls:
            response = self.client.get(elem['url'])
            self.assertEqual(response.status_code,elem['status'])
            response = self.client.get(elem['url'],follow=True)
            self.assertEqual(response.template.name, elem['template'])
```

Lancez une première fois vos tests pour bien vérifier qu'ils échouent. Ensuite, supprimez la ligne suivante de votre fichier `agenda/urls.py` :

```
url(r'^accounts/login/$', login),
```

Relancez vos tests.

> Relancer ses tests entre chaque modification, même si l'on sait pertinemment qu'ils vont échouer, permet de vérifier que l'on n'est pas en train de casser une autre partie de son application.

Ajoutez l'URL au fichier `/agenda/usermanagement/urls.py` :

```
url(r'^login/$', login),
```

Bien entendu, relancez ensuite vos tests. Avez-vous remarqué la dernière ligne de la sortie du terminal ?

```
NameError: name 'login' is not defined
```

En effet, les fonctions `login` et `logout` de `django.contrib.auth.views` sont importées dans `agenda/urls.py`, mais pas dans `agenda/usermanagement/urls.py`. Il faut donc déplacer la ligne suivante et relancer vos tests :

```
from django.contrib.auth.views import login, logout
```

Cette fois, c'est la vue `logout` qui renvoie une erreur d'import. Supprimez la ligne suivante de `agenda/urls.py` :

```
url(r'^accounts/logout/$', logout, {'next_page':'/accounts/login/'}),
```

Puis ajoutez celle-ci dans `agenda/usermanagement/urls.py` :

```
url(r'^logout/$', logout, {'next_page':'/user/login/'}),
```

Finalement, il faut déplacer l'URL :

#agenda.urls.py

```
(r'^accounts/profile/$',login_required(
TemplateView.as_view(template_name="registration/profile.html"))),
```

vers

#agenda.usermanagement.urls.py

```
(r'^profile/$',login_required(
TemplateView.as_view(template_name="registration/profile.html"))),
```

N'oubliez pas de déplacer également l'`import` de `login_required` :

```
from django.contrib.auth.decorators import login_required
```

Si vous lancez vos tests maintenant, vous allez déclencher une erreur à l'URL `/user/profile/` sur la fonction `test_public`. En effet, sans paramétrage supplémentaire, Django redirige automatiquement les pages non autorisées vers `/accounts/login/`, qui n'existe plus dans votre nouvelle configuration.

Pour retrouver un comportement normal, il vous suffit de modifier le `settings.py` de votre application et d'indiquer l'URL de login de votre site :

```
LOGIN_URL = "/user/login/"
```

Pour la même raison, le test `test_login` échouera également. Il vous faut ajouter dans `settings.py` :

```
LOGIN_REDIRECT_URL = "/user/profile/"
```

Vous pouvez alors lancer vos tests avec succès.

#le fichier test.py complet

```
from django.test import TestCase
from django.test import Client
from django.contrib.auth.models import User
class StatusTest(TestCase):
    def setUp(self):
        self.client = Client()

    def test_public(self):
        urls = [{'url':'/user/login/',
                 'template':'registration/login.html',
                 'status':200
                },
                {'url':'/user/logout/',
                 'template':'registration/login.html',
                 'status':302},
                 {'url':'/user/profile/',
                  'template':'registration/login.html',
                  'status':302}
                    ]
        for elem in urls:
            response = self.client.get(elem['url'])
            self.assertEqual(response.status_code,elem['status'])
            response = self.client.get(elem['url'],follow=True)
            self.assertEqual(response.template.name, elem['template'])
```

```
    def test_create_user(self):
         response = self.client.post('/user/create_account/',{
                 'username':'john',
                 'password1':'trytoguess',
                 'password2':'trytoguess'
                 },follow=True)

         self.assertEqual(response.template.name,'user/succes.html')
         user = User.objects.get(username="john")
         self.assertEqual(user.username,"john")

    def test_login(self):
        self.test_create_user()
        response = self.client.post('/user/login/',{
                'username':'john',
                'password':'trytoguess'
                },follow=True)
        self.assertEqual(response.template.name,'registration/profile.html')
```

Modifier les templates

Il reste deux problèmes avec les templates que vous avez écrits.

- Certains sont dans le répertoire `registration` et d'autres dans le répertoire `user`, ce qui n'est pas très logique.
- Vos templates sont à la racine de votre projet. Si un autre développeur insère votre application dans son projet, il devra créer ses propres templates. Une bonne idée pourrait être de lui donner des templates de base dont il pourra s'inspirer.

Nous allons résoudre ces deux problèmes simplement en incluant vos templates dans votre application, ce qui la rendra plus portable. Aucune configuration particulière n'est nécessaire pour faire fonctionner les templates dans l'application. Ils sont appelés si aucun template n'existe au niveau du projet. Ainsi, quand vous réutiliserez l'application `usermanagement`, vous n'aurez qu'à écrire vos propres fichiers dans le répertoire `templates` de votre projet pour que ceux présents dans l'application soient immédiatement ignorés.

Modifiez tout d'abord vos tests pour unifier vos URL en remplaçant toutes les références au dossier `registration` par `user` :

```
    def test_public(self):
        urls = [{'url':'/user/login/',
                 'template':'user/login.html',
                 'status':200
                },
```

```
            {'url':'/user/logout/',
             'template':'user/login.html',
             'status':302},
             {'url':'/user/profile/',
              'template':'user/login.html',
              'status':302}
                ]
        for elem in urls:
            response = self.client.get(elem['url'])
            self.assertEqual(response.status_code,elem['status'])
            response = self.client.get(elem['url'],follow=True)
            self.assertEqual(response.template.name, elem['template'])
```

Encore une fois, vérifiez bien que vos tests échouent – c'est une pratique utile de toujours lancer vos tests *avant* d'implémenter leur réalisation afin de valider qu'ils sont bien écrits. Il vous faut ensuite créer le dossier `agenda/usermanagement/templates/usermanagement/user/` et y déplacer tous les templates présents dans `agenda/templates/registration/` et `agenda/templates/user/`.

Lorsque vous lancerez vos tests, vous vous apercevrez que ceux de l'URL `profile` échouent. Si vous regardez le fichier `agenda/usermanagement/urls.py`, vous trouverez rapidement la réponse. En effet, dans ce fichier, vous utilisez `TemplateView.as_view` qui prend en argument un nom de template, lequel pointe toujours sur `registration`. Modifiez donc :

```
url(r'^profile/$',login_required(
TemplateView.as_view(template_name="registration/profile.html"))),
```

en

```
url(r'^profile/$',login_required( TemplateView.as_view(template_name="user/
profile.html"))),
```

L'URL `login` échouera également car, sans configuration particulière, cette fonction va chercher le template `registration/login.html`. Modifiez simplement :

```
url(r'^login/$', login),
```

en

```
url(r'^login/$', login,{'template_name':'user/login.html'}),
```

Cette dernière instruction va indiquer à la vue `login` de charger le template `user/login.html` et non plus `registration/login.html`. Vous pouvez dès lors relancer vos tests et constater qu'ils fonctionnent désormais tous.

Rev 11

L'application de gestion des utilisateurs réutilisable.

Conclusion

Vous venez de terminer votre première vraie application portable pour Django. Pour valider sa portabilité, vous pouvez, par exemple, la « brancher » sur le projet de tracker que vous avez commencé au début de ce livre. Il vous suffira de l'ajouter aux `INSTALLED_APPS` de votre projet, de renseigner `LOGIN_URL` et `LOGIN_REDIRECT_URL`, et d'entourer d'un `login_required` les URL que vous voulez protéger des utilisateurs non connectés. De plus, vous pourrez créer des templates plus riches que ceux élaborés jusqu'à présent puisque les templates définis au niveau de l'application seront écrasés automatiquement par ceux définis au niveau du projet.

Les tests que vous avez écrits vous permettront de vérifier la bonne intégration de l'ensemble, et vous pourrez continuer à développer cette application en lui ajoutant des fonctionnalités tout en validant que le reste fonctionne toujours normalement.

5

Création d'un événement

Vous disposez à présent d'une application d'authentification fonctionnelle ; vous pouvez penser librement à l'application principale. Avant tout, laissons les utilisateurs créer leurs propres événements.

Dans le précédent chapitre, vous avez utilisé les formulaires de Django pour créer et connecter un utilisateur, mais vous n'avez jamais eu besoin de créer de toutes pièces un formulaire. C'est ce que vous allez faire maintenant. Toutefois, comme le formulaire de création d'un événement repose sur un modèle que vous avez défini dans le fichier `models.py`, vous allez profiter de la puissance de Django, qui va automatiquement créer pour vous les bases d'un formulaire fonctionnel.

Les ModelForm

Les `ModelForm` sont, comme leur nom l'indique, des formulaires basés sur des modèles. Voici comment les utiliser :

1. ajouter au sein de votre application un fichier `forms.py` destiné à recueillir tous les formulaires que vous créez ;
2. importer `ModelFormdepuis` `django.forms` ;
3. créer une classe héritant de `ModelForm` ;

4 définir la classe `Meta` dans laquelle vous indiquez le modèle auquel vous faites référence ;
5 utiliser votre formulaire dans vos vues, de la même manière que le formulaire de création d'utilisateur.

Comme vous le voyez, c'est enfantin. Django va se charger pour vous de définir le type des champs, de valider les informations renvoyées par l'utilisateur et, bien entendu, de la sauvegarde dans la base de données.

Création du formulaire

Créez dans le dossier `personal_calendar` le fichier `forms.py` pour y stocker vos formulaires.

```
from django.forms import ModelForm
from personal_calendar.models import Evenement

class EventForm(ModelForm):
    class Meta:
        model = Evenement
```

Vous héritez de `ModelForm` et créez une nouvelle classe qui en hérite. L'attribut `model` de sa classe `Meta` servira à Django pour créer automatiquement le formulaire. Pour inclure ce formulaire sur votre site, suivez le fonctionnement habituel : créez une nouvelle URL dans `personal_calendar/urls.py` – ce fichier n'existe pas encore, il vous faut le créer –, par exemple :

Création du fichier urls.py

```
from django.conf.urls.defaults import patterns, include, url
from views import create
urlpatterns = patterns('',
            url(r'^create/$',create,
))
```

Référencez ces nouvelles URL dans le fichier `urls.py` du dossier `agenda` :

```
    url(r'^agenda/', include('personal_calendar.urls')),
```

Puis créez la vue correspondante :

```
from forms import EventForm
from django.shortcuts import render, HttpResponseRedirect

def create(request):
    if request.method == "POST":
        form = EventForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/agenda/create/')
    else :
        form = EventForm()
    return render(request,'personal_calendar/event/create.html',{'form':form})
```

Comme vous le voyez, la vue que vous venez d'écrire est identique à la vue de création d'utilisateur que vous avez déjà conçue. Créez ensuite le fichier HTML du formulaire (encore une fois, un template que vous connaissez déjà) :

```
<form action="" method="POST">
{% csrf_token %}
{{form}}
<input type="submit" value="Créer" />
</form>
```

Il vous suffit alors d'afficher le formulaire dans votre navigateur (voir figure 5-1).

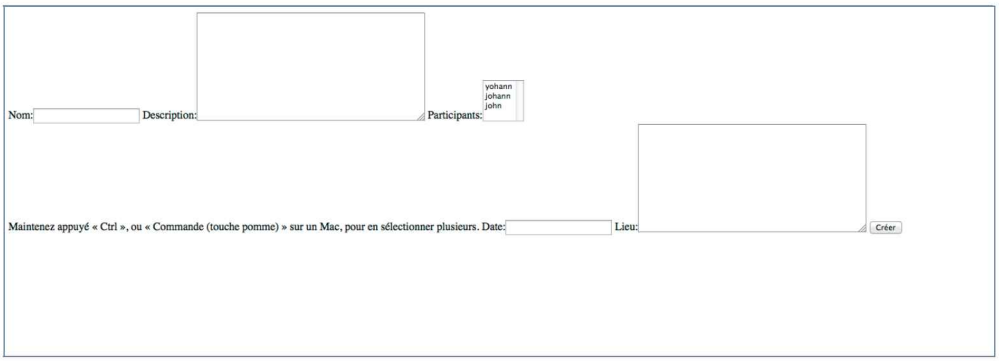


Figure 5–1 Le formulaire de création d'événement

Si vous tentez dès maintenant de créer un nouvel événement, Django va relever une erreur (voir figure 5-2).

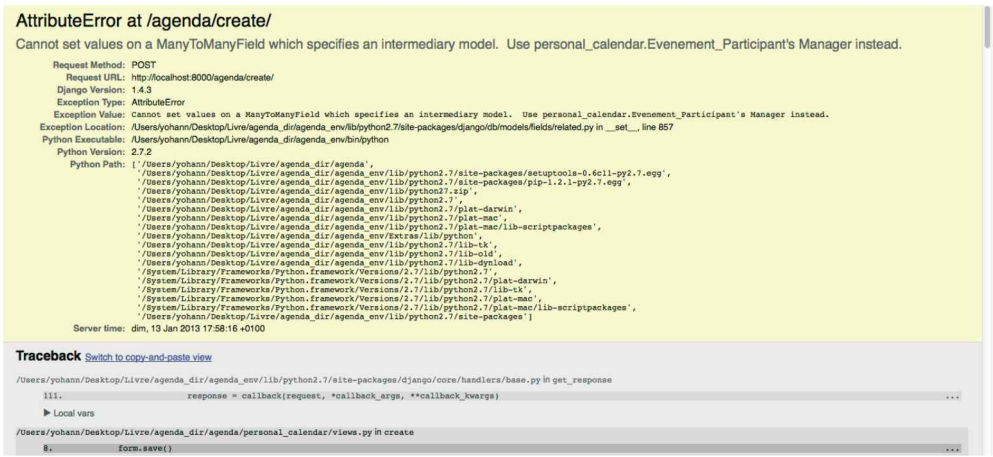


Figure 5–2
Erreur many-to-many

En effet, votre modèle `Evenement` est un peu particulier ; il possède une clé étrangère vers un objet `User` au travers de la table `Evenement_Participant`. Django ne sait pas comment gérer ce type de situation. Néanmoins, comme ce champ n'est pas obligatoire pour l'enregistrement de votre modèle `Evenement`, il suffit de l'exclure du formulaire. Pour ce faire, deux possibilités s'offrent à vous :

- soit indiquer les champs que vous souhaitez exclure :

```
class EventForm(ModelForm):
    class Meta:
        model = Event
        exclude = ('participants',)
```

  On ajoute l'attribut `exclude` à la classe `Meta` de `EventForm`, qui accepte un tuple.

- soit indiquer les champs que vous souhaitez afficher :

```
class EventForm(ModelForm):
    class Meta:
        model = Event
        fields = ('nom','description','date','lieu')
```

  Ici, c'est l'attribut `fields` qui est utilisé.

Le résultat sera exactement le même dans l'un ou l'autre cas. C'est une pure question de convenance.

Vous pouvez désormais tester l'enregistrement de votre formulaire et vérifier qu'il est fonctionnel. Cependant, il souffre de plusieurs défauts.

- Rien ne permet de savoir si la création est effective. On pourrait renvoyer vers une page indiquant le succès de l'opération, mais cela ajouterait un niveau de complexité pour l'utilisateur, qui devrait alors revenir en arrière pour, par exemple, créer un nouvel événement.
- Ce formulaire est incomplet. Il doit être possible d'ajouter des participants et de définir leur statut facilement.

Rev 12

Un premier formulaire succinct.

Ces deux problèmes vont être corrigés en une seule action.

Présenter l'objet Evenement à l'utilisateur

Après validation du formulaire, il suffit de présenter à l'utilisateur une page où il enregistrera les participants à cet événement. Ces derniers seront ajoutés via un formulaire basé sur le modèle `Evenement_Participant`. Évidemment, il faut pouvoir enregistrer plusieurs participants à la fois. Ce formulaire aura donc la particularité de présenter une collection de formulaires dont l'attribut `evenement` correspond à l'événement que vous venez de créer.

Ce fonctionnement se fait en trois étapes.

1. Créer une page affichant les données d'un événement.
2. Créer un ensemble de formulaires pour le modèle `Evenement_Participant`.
3. Instancier les formulaires correspondant à l'événement présenté.

Il faut créer une nouvelle vue qui va prendre en paramètre l'identifiant unique d'un événement, chercher cet événement en base de données et le retourner à un nouveau template. Au début du fichier `personal_calendar/views.py`, importez la classe `Evenement` de vos modèles :

```
from models import Evenement
```

Puis créez votre vue comme ceci :

```
def details(request, id):
    event = Evenement.objects.get(pk = id)
    return render(request,'personal_calendar/event/
details.html',{'event':event})
```

Dans `agenda/templates/personal_calendar/event/details.html`, il vous faut ensuite créer le template qui affichera les données de l'événement :

```
<h1>{{event.nom}}</h1>
<p>{{event.description}}</p>
<p>le {{event.date}}</p>
<p>à {{event.lieu}}</p>
```

Ce n'est certes pas la plus belle présentation qu'il vous ait été donné de voir, mais ce sera fonctionnel pour le moment – vous verrez bientôt comment rendre tous vos templates beaucoup plus dynamiques et attrayants.

Enregistrez ensuite l'URL qui servira cette nouvelle vue :

```
from views import create, details
xs
url(r'^(\d+)/details/$',details),
```

La partie `(\d+)` est une expression régulière, comme vous avez pu en voir dans l'application `tracker`, qui va enregistrer un ou plusieurs (le signe `+`) caractères numériques (le signe `d`) à renvoyer en paramètre à la vue. Testez cette nouvelle vue et constatez son bon fonctionnement.

Rev 13

Une page `details` pour les événements.

La seconde étape va consister à faire en sorte qu'à l'enregistrement d'un nouvel événement, l'utilisateur soit renvoyé sur la page `details` de l'événement. Il faut donc modifier très légèrement la vue `create` :

```
def create(request):
    if request.method == "POST":
        form = EventForm(request.POST)
        if form.is_valid():
            event = form.save()
            return HttpResponseRedirect('/agenda/%i/details/'%event.pk)
    else:
        form = EventForm()
    return render(request,'personal_calendar/event/create.html',{'form':form})
```

Seules les lignes sous `if form.is_valid()` nécessitent d'être modifiées, car c'est seulement dans le cas où le formulaire est valide qu'il faut rediriger l'utilisateur vers la page `details` du nouvel événement. L'astuce ici est d'enregistrer le résultat de `form.save()`

dans la variable `event`. En effet, `form.save()` renvoie l'objet `Evenement` qu'il vient d'enregistrer dans la base de données. Pour vous en convaincre, vous pouvez le tester en écrivant juste après la ligne :

```
event = form.save()
print event.__class__
```

La fonction ajoutée crée l'URL vers laquelle il faut renvoyer l'utilisateur.

Ajouter des participants

La suite du développement de l'application consiste à inscrire de nouveaux participants à l'événement que l'on vient de créer. Encore une fois, il faut procéder par étapes.

1. Créer un formulaire pour le modèle `Evenement_Participant`.
2. Lier ce formulaire à un événement.

La création du formulaire suit naturellement le processus que vous avez suivi jusqu'à maintenant. Dans le fichier `agenda/personal_calendar/forms.py`, écrivez :

```
class Evenement_ParticipantForm(ModelForm):
    class Meta:
        model = Evenement_Participant
```

N'oubliez pas d'inclure en début de fichier un `import` du modèle `Evenement_Participant` :

```
from personal_calendar.models import Evenement_Participant
```

Il vous faut ensuite ajouter ce formulaire et sa mécanique de validation à la vue `details` :

```
def details(request,id):
    if request.method == "POST":
        form = Evenement_ParticipantForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/agenda/%s/details/' % event)
    else:
        form = Evenement_ParticipantForm()
    event = Evenement.objects.get(pk = id)
    return render(request,'personal_calendar/event/details.html',{
            'event':event,
            'form':form})
```

Remarquez que c'est une vue honteusement copiée sur la vue `create` et qu'il faudrait certainement factoriser cette partie. Toutefois, ces vues seront bientôt remplacées par d'autres plus simples. Nous resterons donc sur ce principe afin de bien saisir le fonctionnement parfois complexe des formulaires.

Remarquez également que nous avons utilisé une substitution de type chaîne de caractères au lieu d'un type entier comme sur la vue `create`. Dans cette dernière en effet, l'identifiant unique de l'événement est fourni par l'ORM de Django ; il a donc conservé son type. En revanche, dans la seconde fonction, l'identifiant unique est fourni par l'URL et est interprété par Django comme une chaîne de caractères.

Ensuite, modifiez le template `details.html` en ajoutant à la fin les lignes suivantes :

```
<form action="" method="POST">
{% csrf_token %}
{{form.as_p}}
<input type="submit" value="Créer" />
</form>
```

Visualisez une page de détail, par exemple `http://localhost:8000/event/1/details/`. Le nouveau formulaire est disponible. Vous pouvez même commencer à y insérer de nouveaux participants. Seulement, il reste plusieurs point gênants dont le principal est sans doute que l'on ne voit pas les participants à l'événement sur la page `details.html`. Vous devez également choisir l'événement qui va recevoir un nouveau participant, celui que vous consultez n'étant pas automatiquement sélectionné.

C'est assez simple à corriger, moyennant une petite astuce. L'objet que vous affichez dans le template `details` est de la classe `Evenement`. Regardez comment il se présente dans le fichier modèle :

```
    nom = models.CharField(max_length = 250)
    description = models.TextField()
    participants = models.ManyToManyField(User,through =
    "Evenement_Participant", null=True,blank=True)
    date = models.DateTimeField()
    lieu = models.TextField()
```

Si vous souhaitez afficher les participants, vous devriez afficher le champ `participant`. Dans votre template, il semblerait logique d'écrire :

```
{% for participant in event.participants %}
{{participant}}
{% endfor %}
```

`event.participants` étant l'équivalent de `event.participants.all()` au niveau du template.

Cependant, si vous testez cette vue, Django vous renvoie une erreur.

Vous serez confronté à cette erreur dès que vous voudrez afficher un champ de modèle correspondant à une liste d'objets d'un autre modèle, en d'autres termes dès que vous voudrez afficher un champ de type *many-to-many*. Les champs de ce type renvoient un *manager*, c'est-à-dire une classe qui se charge des transactions avec l'ORM, en coulisses. Nous verrons en détail dans les chapitres suivants ce que sont les managers, comment les utiliser et comment créer les vôtres.

> **TECHNIQUE Pourquoi Django renvoie-t-il un manager ?**
>
> Django se positionne comme le framework des perfectionnistes avec des délais ; il se doit donc d'être performant. Et l'une des clés de la performance est d'optimiser les requêtes en base de données. Dans cette optique, à chaque fois que vous faites appel à un manager, Django garde en mémoire votre requête, mais ne l'effectue pas immédiatement : elle n'est exécutée qu'au moment où vous utilisez l'objet ou les objets qu'elle doit retourner. Ainsi, quand vous faites appel à un objet de type `Evenement`, Django ne fait pas de requêtes sur les objets `User` contenus dans l'objet, mais instancie un manager qui contactera la base de données au moment où vous le souhaiterez. De plus, lorsque vous appelez réellement les objets, en utilisant une boucle dans votre code ou en les affichant dans votre template par exemple, Django ne charge pas les objets en relation, mais rend disponibles de nouveaux managers pour ces objets liés. Un fonctionnement différent aurait des conséquences désastreuses sur les performances : imaginez un objet référençant trois autres modèles, qui eux-mêmes référenceraient trois autres modèles, et ainsi de suite. Le nombre de requêtes que devrait traiter Django serait alors incalculable et les performances de votre application en seraient largement diminuées.

Pour le moment, sachez que vous utilisez déjà les managers quand vous vous servez de l'ORM. Par exemple, quand vous écrivez :

```
Evenement.objects.get(pk = event.id)
```

vous utilisez la fonction `get` du manager `objects` de la classe `Event` ; `objects` est un manager défini par défaut à la création d'une classe héritant de `models.Model`.

Maintenant que vous savez, d'après l'erreur que Django vous a renvoyée, que `event.participants` est un manager, il vous suffit de l'utiliser comme tel. Vous pourriez, par exemple, écrire :

```
event.participants.all()
```

Cependant, comme vous ne pouvez pas utiliser de fonction dans les templates, du moins pas directement, Django vous permet d'employer la syntaxe suivante :

```
{% for participant in event.participants.all %}
{{ participant }}
{% endfor %}
```

Cette fois, le résultat est bien plus satisfaisant. Vous affichez tous les utilisateurs inscrits à cet événement.

Vous voudrez sans doute afficher également le statut de chaque utilisateur :

```
{{ participant.status }}
```

Là, en revanche, rien ne s'affiche. La raison de cet échec est simple : le modèle `Evenement` est lié à un objet `User` à travers un objet de type `Evenement_Participant`. Quand vous affichez `event.participants.all`, Django vous renvoie les objets de type `User` liés à votre événement. Pour présenter le statut d'un utilisateur, vous avez besoin d'afficher les objets `Evenement_Participant` liés à votre objet `Evenement`. Le problème ici est que les `Evenement_Participant` n'apparaissent pas sur l'objet `Evenement`. Dans le fichier `models.py`, c'est l'objet `Evenement_Participant` qui référence l'objet `Evenement`.

Pourtant, Django vous offre la possibilité d'accéder aux objets qui référencent l'objet que vous manipulez. Pour bien comprendre ce point fondamental du fonctionnement de l'ORM, reprenez votre objet `Evenement` : il référence des objets `User` et vous pouvez accéder aux objets `User` via `participants.all` car `participants` est l'un des champs de l'objet `Evenement`. Du point de vue de l'objet `User`, aucun champ ne référence d'objet `Evenement`. Pourtant, vous pouvez y accéder avec `_set`. Reprenez votre template :

```
{% for participant in event.participants.all %}
{{participant}}
{% endfor %}
```

Si vous souhaitez afficher tous les événements auxquels participent les utilisateurs inscrits à l'événement courant, modifiez votre template :

```
{% for participant in event.participants.all %}
{{participant.event_set.all}}
{% endfor %}
```

Ici, `event_set` est un manager créé par Django pour atteindre les événements de l'utilisateur. Pour afficher les objets de `Evenement_Participant` liés à votre objet `Evenement`, il vous suffit donc d'écrire :

```
{% for participant in event.evenement_participant_set.all %}
{{participant.participant}} {{participant.status}}
{% endfor %}
```

En rafraîchissant la page, vous constatez que, cette fois, le statut de chaque utilisateur s'affiche correctement, même si vous auriez sans doute préféré voir « hôte » au lieu de « 0 ». Pour accéder à la seconde valeur du champ `status`, il vous suffit d'utiliser la fonction `get_status_display`, qui est automatiquement créée par Django pour vous donner l'accès au label de votre champ :

```
{% for participant in event.evenement_participant_set.all %}
{{participant.participant}} {{participant.get_status_display}}
{% endfor %}
```

Améliorations et finitions

La fonctionnalité de création d'événement est désormais implémentée. Il est temps de s'occuper des détails et d'améliorer l'ergonomie.

Les contraintes

Peut-être avez-vous constaté que le formulaire d'ajout de participants vous autorise à enregistrer plusieurs fois le même utilisateur pour un événement donné. Dans le cadre d'un agenda, ce n'est pas logique. Nous allons remédier à cela.

Cette contrainte d'unicité vous sera nécessaire à plusieurs endroits de votre application, pas seulement dans le formulaire d'ajout de participants. C'est pourquoi vous allez la créer directement dans vos modèles. Ainsi, elle sera automatiquement effective pour toute l'application. De la même façon, vous pouvez faire en sorte que le nom de l'événement soit unique lui aussi, pour retrouver facilement un événement à partir de son nom.

L'attribut unique pour la contrainte d'unicité

Toutes les contraintes d'unicité sont gérées par la base de données que vous utilisez et non pas par Django lui-même, le principe de l'ORM de Django étant de déléguer le plus possible à la base de données afin de garantir les meilleures performances. Il

vous suffit de positionner à `True` l'attribut `unique` d'un champ. Par exemple, si vous souhaitez que le nom d'un événement soit unique, modifiez le champ `nom` du modèle `Evenement` de la façon suivante :

```
nom = models.CharField(max_length = 250, unique=True)
```

Comme cette directive est prise en compte au niveau de la base de données, il vous faut également modifier le schéma de votre base pour que cette modification soit pleinement effective. Habituellement, cette modification en base de données est assez pénible : vous devez utiliser des outils tels que `python manage.py dbshell` et lancer des commandes SQL pour modifier votre base à la main. Heureusement, vous avez déjà installé `south`, dont le but est justement de vous éviter cette démarche.

Tapez la commande suivante dans le terminal :

```
python manage.py schemamigration personal_calendar --auto
```

`south` vous indique alors qu'il a bien détecté les modifications que vous vouliez effectuer :

```
+ Added unique constraint for ['nom'] on personal_calendar.Evenement
```

et vous invite à faire la migration avec la commande :

```
python manage.py migrate personal_calendar
```

Avant toute chose, assurez-vous qu'il n'y a pas plusieurs événements qui portent le même nom car, sinon, la migration échouera. Supprimez de la base les événements problématiques. Toutefois, comme nous sommes en cours de développement et qu'aucun événement n'a vraiment d'importance, vous pouvez utiliser une commande très pratique de Django : `python manage.py reset`. Elle prend en argument le nom de l'une de vos applications et détruit tous les enregistrements présents dans la base de données associée. Pour vous assurer que la migration se passe bien, vous pouvez donc taper `python manage.py reset personal_calendar`, puis effectuer votre migration sans risque. Désormais, vous ne pouvez plus créer deux événements de même nom.

L'attribut unique_together

Le second type de contrainte que vous allez mettre en place concerne les deux champs `evenement` et `participant` du modèle `Evenement_Participant`. Cette fois, vous n'utiliserez pas l'attribut `unique` – il ne fonctionne que pour un seul champ –, mais `unique_together`.

Vous allez créer votre nouvelle contrainte à l'aide de la classe `Meta` de votre modèle, qui fonctionne de la même manière que la classe `Meta` des formulaires. Il vous suffit donc d'écrire dans le modèle `Evenement_Participant` :

```
class Meta:
        unique_together = ("evenement","participant")
```

Il vous faut ensuite indiquer votre modification à `south` et l'effectuer au niveau de la base de données :

```
python manage.py schemamigration personal_calendar --auto
python manage.py migrate personal_calendar
```

Rev 4

Les premières migrations.

Rendre intelligent le formulaire

Maintenant que les contraintes sont effectives au niveau de la base de données, il vous faut encore simplifier le remplissage des formulaires. Par exemple, lorsque vous présentez le formulaire d'ajout d'un participant dans la page d'un événement, l'utilisateur ne devrait pas avoir à sélectionner lui-même l'événement. Ce champ ne devrait même pas être affiché.

De la même manière, puisqu'un événement ne doit pas enregistrer plusieurs fois le même utilisateur, vous devriez supprimer ce dernier de la liste des choix possibles et par là même le formulaire d'ajout de participant si tous les utilisateurs ont déjà été sélectionnés. Vous pouvez également proposer un bouton pour supprimer un participant d'un événement. Les modifications de ce type rendent l'utilisation de votre application beaucoup plus conviviale pour l'utilisateur.

Présélectionner l'événement

Django préremplit les champs de formulaire via l'attribut `initial`, qui prend en argument un dictionnaire. Les clés de ce dernier sont les champs que vous souhaitez préremplir et les valeurs sont, bien entendu, celles de ces champs. Dans le cas présent, vous pouvez écrire dans votre vue :

```
form = Evenement_ParticipantForm(initial = {'evenement':event})
```

Votre vue `details` ressemblera donc à ceci :

```
def details(request, id):
    event = Evenement.objects.get(pk = id)
    if request.method == "POST":
        form = Evenement_ParticipantForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/agenda/%s/details/' % id)
    else:
        form = Evenement_ParticipantForm(initial = {'evenement':event})
    return render(request,'personal_calendar/event/details.html',{
            'event':event,
            'form':form})
```

Si vous la testez maintenant, vous constaterez que le champ `evenement` est bien placé sur la bonne valeur.

Supprimer les choix inutiles

Le formulaire d'ajout de participants est créé automatiquement par Django. Les champs `evenement` et `participant` sont liés respectivement aux modèles `Evenement` et `User`. Pour établir la liste de choix des participants, Django fait une *queryset* de ce type :

```
participants = User.objects.all()
```

Il vous suffit donc de modifier cette queryset pour filtrer les utilisateurs qui participent déjà à cet événement. Récupérez ces derniers via le champ `participant` de l'objet `Evenement` qui, comme vous l'avez vu précédemment, est un manager. Vous devez utiliser l'une des méthodes de ce manager (`all`, `filter`, `get`...). Ici, vous souhaitez récupérer tous les utilisateurs ; la méthode `all` est donc la plus indiquée :

```
participants = event.participants.all()
```

Il vous faut ensuite exclure le résultat de cette queryset des choix présentés par le champ `participant` du formulaire `Evenement_ParticipantForm`. Le plus simple est donc de récupérer une liste des identifiants uniques des participants et de l'exclure des choix possibles :

```
participants = [user.pk for user in event.participants.all()]
form.fields['participant'].queryset=User.objects.exclude(
pk__in = participants)
```

N'oubliez pas d'importer le modèle `User` en début de fichier :

```
from django.contrib.auth.models import User
```

Sur la première ligne, vous récupérez les identifiants des participants à l'événement. Sur la seconde ligne, vous modifiez l'attribut `queryset` du champ `participant`de l'objet `form` en excluant la liste des utilisateurs déjà présents. Le filtre `__in` compare un attribut de votre objet à une liste : ici, le champ `pk` (l'identifiant unique) à la liste des identifiants que vous avez récupérés plus haut.

Votre fonction ressemble maintenant à ceci :

```
def details(request, id):
    event = Evenement.objects.get(pk = id)
    if request.method == "POST":
        form = Evenement_ParticipantForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/agenda/%s/details/' % id)
    else:
        form = Evenement_ParticipantForm(initial = {'evenement':event})
        participants = [user.pk for user in event.participants.all()]
        form.fields['participant'].queryset=User.objects.exclude(
            pk__in = participants)
    return render(request,'personal_calendar/event/details.html',{
            'event':event,
            'form':form})
```

Supprimer un participant

La suppression d'un objet de la base de données se fait grâce à la méthode `delete` disponible automatiquement pour chaque modèle. Pour supprimer un participant, il vous suffit donc d'utiliser cette méthode sur un objet de type `Evenement_Participant`, via un nouveau formulaire très simple dans votre template :

```
{% for participant in event.evenement_participant_set.all %}
<p>{{participant.participant}} {{participant.get_status_display}}
<form action="/agenda/{{event.id}}/participant/{{participant.id}}/delete/"
method="POST">
  {% csrf_token %}
  <input type="submit" value="Supprimer" />
</form>
</p>
{% endfor %}
```

L'URL `/event/{{event.id}}/participant/{{participant.id}}/delete/` est composée logiquement et donne toutes les informations dont la vue que vous allez écrire a besoin :

- `{{event.id}}` : l'identifiant unique de votre événement ;
- `{{participant.id}}` : l'identifiant unique de votre participant.

Dans le fichier `agenda/personal_calendar/urls.py`, ajoutez votre nouvelle URL :

```
from views import create, details, delete_participant
url(r'^(\d+)/participant/(\d+)/delete/$', delete_participant),
```

Vous pouvez maintenant créer votre nouvelle vue. Tout d'abord, il vous faut vérifier que la requête est bien effectuée avec le verbe `POST`, afin d'éviter de supprimer un participant par inadvertance, simplement en copiant-collant une URL :

```
def delete_participant(request, id, participant):
    if request.method == "POST":
```

Il vous suffit ensuite de récupérer l'événement et le participant dont vous avez besoin pour identifier avec certitude l'objet `Evenement_Participant` que vous devez supprimer – ces deux champs ayant la contrainte `unique_together`, vous êtes certain de retrouver toujours un objet unique :

```
evenement = Evenement.objects.get(pk = id)
participant = User.objects.get(pk = participant)
```

Recherchez ensuite l'objet `Evenement_Participant` que vous devez supprimer :

```
a_supprimer = Evenement_Participant.objects.get(
                        evenement = evenement,
                        participant = participant
                        )
a_supprimer.delete()
```

La dernière chose qui vous reste à faire est de rediriger l'utilisateur vers l'événement :

```
return HttpResponseRedirect('/agenda/%s/details/' % event)
```

Voici donc la fonction dans son ensemble :

```
from models import Evenement, Evenement_Participant

def delete_participant(request, id, participant):
```

```
    if request.method == "POST":
        evenement = Evenement.objects.get(pk = id)
        participant = User.objects.get(pk = participant)
        a_supprimer = Evenement_Participant.objects.get(
                      evenement = evenement,
                      participant = participant
                      )
        a_supprimer.delete()
    return HttpResponseRedirect('/agenda/%s/details/' % id)
```

N'afficher que les champs utiles

Votre formulaire commence à être vraiment simple et facile d'utilisation. Un dernier point reste pourtant à modifier. À l'affichage de votre formulaire, même si le champ `evenement` est positionné sur l'événement en cours, il reste affiché. Il faut bien entendu supprimer ce champ de l'affichage, mais sans perdre la création automatique du formulaire. Si, plus tard, vous ajoutez un champ à votre modèle, il faut que le formulaire soit automatiquement modifié afin de garder une logique DRY : une modification ne doit se faire qu'à un seul endroit de votre application et se répercuter partout ailleurs.

La première chose à faire est d'indiquer à Django que le champ `evenement` doit être de type `hidden`, en modifiant son attribut `widget`. Ce dernier est habituellement automatiquement choisi par Django en fonction du champ modèle qu'il représente. Pour le moment, il s'agit d'un champ de type `ModelChoiceField` auquel est associé par défaut un widget de type `Select`. Cependant, comme vous avez défini sa valeur avec le paramètre initial, vous pouvez utiliser le widget `HiddenInput` qui, comme son nom le laisse supposer, présente un champ caché.

Importez donc le widget `HiddenInput` au début de votre fichier `agenda/personal_calendar/views.py` :

```
from django.forms import HiddenInput
```

Ensuite, vous l'utilisez dans votre fonction `details` :

```
form.fields['evenement'].widget = HiddenInput()
```

Votre fonction complète ressemble désormais à ceci :

```
def details(request, id):
    event = Evenement.objects.get(pk = id)
    if request.method == "POST":
        form = Evenement_ParticipantForm(request.POST)
```

```
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/agenda/%s/details/' % id)
    else:
        form = Evenement_ParticipantForm(initial = {'evenement':event})
        participants = [user.pk for user in event.participants.all()]
        form.fields['participant'].queryset=User.objects.exclude(
            pk__in = participants)
        form.fields['evenement'].widget = HiddenInput()
    return render(request,'personal_calendar/event/details.html',{
            'event':event,
            'form':form})
```

Rev 15

La vue `details` terminée.

L'interface CRUD

Votre application est maintenant bien avancée. Vous ne vous en rendez peut-être pas encore compte, mais vous avez posé un ensemble de bases qui vous permettront d'ajouter très simplement de nouvelles fonctionnalités. Il ne manque plus grand-chose au cœur de votre application.

Avant d'utiliser concrètement votre agenda, ajoutons la possibilité de lister tous les événements et d'en supprimer un de la liste.

Lister tous les événements

Créez tout d'abord une nouvelle URL :

```
url(r'^liste/$',liste),
```

Importez en début de fichier la fonction `liste` que vous allez écrire :

```
from views import liste
```

La vue `listing` s'explique d'elle-même :

```
def liste(request):
    events = Evenement.objects.all()
    return render(request,'personal_calendar/event/liste.html',{"events" :
events})
```

Créez ensuite le template `personal_calendar/event/liste.html` par exemple :

```
<ul>
{% for event in events %}
<li>
   <h3>{{event.nom}}</h3>
      <p>le {{event.date}}</p>
      <p>à {{event.lieu}}</p>
{% endfor %}
</ul>
<p>
<a href="/agenda/create/">Créer un nouvel événement</a>
</p>
```

Tout ceci devrait vous être familier maintenant. La seule chose qu'il reste à faire est d'indiquer, pour chaque événement, une URL qui pointera naturellement vers la vue `details`. Plutôt que d'écrire par exemple :

```
<a href="/event/{{event.id}}/details/">{{event.nom}}</a>
```

vous pouvez définir dans les modèles une URL qui identifiera un objet `Evenement` unique. Cette fonction sera ensuite utilisée à de nombreux endroits de votre application, par exemple dans les flux RSS automatiquement produits par Django. Pour définir cette fonction, il suffit d'ajouter les lignes suivantes dans votre modèle `Evenement` :

```
    def get_absolute_url(self):
        return "/agenda/%s/details" % self.id
```

```
<ul>
{% for event in events %}
<li>
   <h3><a href="{{event.get_absolute_url}}">{{event.nom}}</a></h3>
      <p>le {{event.date}}</p>
      <p>à {{event.lieu}}</p>
{% endfor %}
</ul>
<p>
<a href="/agenda/create/">Créer un nouvel événement</a>
</p>
```

Dès que vous souhaiterez vous référer à l'URL d'un événement, vous emploierez cette fonction.

Supprimer un événement

Pour donner la possibilité de détruire un objet `Event`, utilisez le même principe que pour la destruction d'un participant.

1. Créez une nouvelle URL :

```
url(r'^(\d+)/delete/$',delete),
```

2. Importez la fonction `delete` dans vos URL :

```
from views import delete
```

3. Créez la fonction `delete` dans vos vues :

```
def delete(request, id):
    if request.method == "POST":
        evenement = Evenement.objects.get(pk = id)
        evenement.delete()
    return HttpResponseRedirect('/agenda/liste/')
```

4. Dans vos templates, créez un formulaire pour supprimer l'événement. Par exemple, dans le template `details.html` :

```
<form action="/agenda/{{event.id}}/delete/"
      method="POST">
  {% csrf_token %}
  <input type="submit" value="Supprimer" />
</form>
```

Modifier un événement

Enfin, il faut pouvoir modifier un événement existant, par exemple pour mettre à jour sa description ou changer sa date. Là aussi, vous allez faire tout cela très rapidement, en reprenant le code déjà écrit pour la création d'un événement. Comme d'habitude, cela commence par les URL :

```
from views import update
url(r'^(\d+)/update/$',update),
```

La vue `update` est semblable à la vue `create` :

```
def update(request, id):
    event = Evenement.objects.get(pk = id)
    if request.method == "POST":
        print request.POST
```

```
        form = EventForm(request.POST, instance=event)
        if form.is_valid():
            a = form.save()
            return HttpResponseRedirect('/agenda/%i/details/' % a.pk)
    else:
        form = EventForm(instance = event)
    return render(request,'personal_calendar/event/create.html',{'form':form})
```

La seule différence avec la vue `create` est l'usage du paramètre `instance` dans la création du formulaire, qui détermine l'objet `Evenement` qui va être modifié.

Ajoutez cette nouvelle URL dans le template `liste` :

```
<a href="/agenda/{{event.id}}/update/">Modifier</a>
```

Conclusion

Vous avez maintenant une application totalement CRUD puisque vous avez implémenté les quatre actions principales : *Create*, *Read*,*Update* et *Delete*.

Tout ceci est écrit avec relativement peu de code. Pour mémoire, voici le code que vous devriez obtenir jusqu'ici ; nous avons condensé l'écriture et ajouté des liens là où ils étaient nécessaires.

Les URL : #agenda/personal_calendar/urls.py

```
from django.conf.urls.defaults import patterns, include, url
from views import create, delete, details, update, liste, delete_participant

urlpatterns = patterns('',
    url(r'^create/$', create),
    url(r'^(\d+)/details/$',details),
    url(r'^liste/$',liste),
    url(r'^(\d+)/delete/$',delete),
    url(r'^(\d+)/participant/(\d+)/delete/$', delete_participant),
    url(r'^(\d+)/update/$',update),
    )
```

Les vues : #agenda/personal_calendar/views.py

```
from forms import EventForm, Evenement_ParticipantForm
from models import Evenement, Evenement_Participant
from django.shortcuts import render, HttpResponseRedirect
from django.contrib.auth.models import User
from django.forms import HiddenInput

def create(request):
    if request.method == "POST":
        form = EventForm(request.POST)
        if form.is_valid():
            event = form.save()
            return HttpResponseRedirect('/agenda/%i/details/'%event.pk)
    else:
        form = EventForm()
    return render(request,'personal_calendar/event/create.html',{'form':form})

def details(request, id):
    event = Evenement.objects.get(pk = id)
    if request.method == "POST":
        form = Evenement_ParticipantForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/agenda/%s/details/' % id)
    else:
        form = Evenement_ParticipantForm(initial = {'evenement':event})
        participants = [user.pk for user in event.participants.all()]
        form.fields['participant'].queryset=User.objects.exclude(
            pk__in = participants)
        form.fields['evenement'].widget = HiddenInput()
    return render(request,'personal_calendar/event/details.html',{
            'event':event,
            'form':form})

def delete_participant(request, id, participant):
    if request.method == "POST":
        evenement = Evenement.objects.get(pk = id)
        participant = User.objects.get(pk = participant)
        a_supprimer = Evenement_Participant.objects.get(
                      evenement = evenement,
                      participant = participant
                      )
        a_supprimer.delete()
    return HttpResponseRedirect('/agenda/%s/details/' % id)

def liste(request):
    events = Evenement.objects.all()
    return render(request,'personal_calendar/event/liste.html',{"events" :
events})
```

```
def delete(request, id):
    if request.method == "POST":
        evenement = Evenement.objects.get(pk = id)
        evenement.delete()
    return HttpResponseRedirect('/agenda/liste/')

def update(request, id):
    event = Evenement.objects.get(pk = id)
    if request.method == "POST":
        print request.POST
        form = EventForm(request.POST, instance=event)
        if form.is_valid():
            a = form.save()
            return HttpResponseRedirect('/agenda/%i/details/' % a.pk)
    else:
        form = EventForm(instance = event)
    return render(request,'personal_calendar/event/create.html',{'form':form})
```

Les formulaires

```
from django.forms import ModelForm
from personal_calendar.models import Evenement
from personal_calendar.models import Evenement_Participant

class EventForm(ModelForm):
    class Meta:
        model = Evenement
        exclude = ('participants', )

class Evenement_ParticipantForm(ModelForm):
    class Meta:
        model = Evenement_Participant
```

Les modèles (sans les docstrings) :#agenda/models.py

```
#-*-coding:utf-8 -*-
from django.db import models
from django.contrib.auth.models import User
#on rend disponible l'objet "user" pour le référencer dans les modèles

class Evenement(models.Model):
    nom = models.CharField(max_length=250, unique=True)
    description = models.TextField()
    participants = models.ManyToManyField(
                        User,
                        through="Evenement_Participant",
```

```
                )
    date = models.DateTimeField()
    lieu = models.TextField()

    def get_absolute_url(self):
        return "/agenda/%s/details" % self.id

class Evenement_Participant(models.Model):
    #notre table de relation
    evenement = models.ForeignKey(Evenement)
    participant = models.ForeignKey(User)
    status_choices = (
                      (0, "hôte"),
                      (1, "invité"),
                      (2, "désisté")
                      )
    status = models.IntegerField(choices=status_choices)

    class Meta:
        unique_together = ("evenement","participant")
```

Le template liste.html

```
<ul>
{% for event in events %}
<li>
   <h3><a href="{{event.get_absolute_url}}">{{event.nom}}</a></h3>
      <p>le {{event.date}}</p>
      <p>à {{event.lieu}}</p>
{% endfor %}
</ul>
<p>
<a href="/agenda/create/">Créer un nouvel événement</a>
</p>
```

Le template create.html

```
<form action="" method="POST">
{% csrf_token %}
{{form}}
<input type="submit" value="Créer" />
</form>
```

Le template details.html

```
<h1>{{event.nom}}</h1>
<p>{{event.description}}</p>
<p>le {{event.date}}</p>
<p>à {{event.lieu}}</p>
{% for participant in event.evenement_participant_set.all %}
{{participant.participant}} {{participant.get_status_display}}
<form action="/agenda/{{event.id}}/participant/{{participant.id}}/delete/"
method="POST">
  {% csrf_token %}
  <input type="submit" value="Supprimer" />
</form>
</p>
{% endfor %}
<form action="" method="POST">
{% csrf_token %}
{{form.as_p}}
<input type="submit" value="Créer" />
</form>

<form action="/agenda/{{event.id}}/delete/"
      method="POST">
  {% csrf_token %}
  <input type="submit" value="Supprimer" />
</form>
<a href="/agenda/{{event.id}}/update/">Modifier</a>
```

Rev 16

Un système CRUD complet.

Comme vous le voyez, c'est très peu de code pour une application qui est maintenant parfaitement fonctionnelle. Et pourtant, vous pourriez encore factoriser bien du code. Par exemple, les formulaires `create` et `update` sont très semblables ; il serait bien plus efficace de n'en faire qu'une seule fonction. Vous pourriez également nommer vos URL pour rendre votre application plus robuste aux changements. Il est aussi possible de renforcer le découplage de votre application via un système de chargement d'URL automatique avec la syntaxe `{% url %}` dans vos templates et *reverse* dans vos vues (voir chapitre 16).

Lancez-vous dans ce type de tâches à titre d'exercice, mais sachez que, très bientôt, le code que vous avez écrit va être intégralement remplacé par une méthode encore plus rapide : les vues génériques. Nous avons déjà évoqué la vue `TemplateView`, mais il y en a beaucoup d'autres qui vont vous aider à développer votre application très rapide-

ment. Cependant, comme pour tous les outils très puissants, si vous n'avez pas une connaissance approfondie des rouages de Django, il est probable que les vues génériques vous fassent perdre plus de temps qu'elles ne vous en feraient gagner.

Avant de traiter les vues génériques, nous allons nous pencher sur les templates. Pour le moment, vous ne les avez utilisés que pour afficher de façon très sommaire les informations renvoyées par vos vues. Voyons maintenant comment les transformer en un véritable site Internet. Il sera donc question, dans le prochain chapitre, de CSS, d'images et de JavaScript.

6

Les templates pour l'affichage

Jusqu'à maintenant, vous vous êtes surtout attardé sur des considérations purement techniques : l'organisation de votre application et la gestion des données utilisateur. Si cette partie est bien entendu la plus importante d'un projet Django et d'un projet web en général, le dialogue avec l'utilisateur et la manière dont il est présenté restent primordiaux pour fidéliser vos visiteurs.

> Nous approfondissons la compréhension du langage des templates au chapitre 12.

Fonctionnement général : découplage et souplesse pour l'intégration avec CSS/JavaScript

Django s'appuie sur quelques assertions en ce qui concerne l'affichage des données aux utilisateurs.

- Tout d'abord, les développeurs de Django considèrent que le travail de concepteur et d'intégrateur est bien différent de celui de développeur. Ainsi, vous devez pouvoir travailler dans un type d'organisation où les développeurs et les intégrateurs travaillent en parallèle.

- La seconde assertion que pose Django est qu'un concepteur et, a fortiori, un intégrateur doivent être en mesure d'éditer des fichiers HTML. Les outils comme Dreamweaver sont donc de fait prohibés.
- Il est volontairement fait en sorte de limiter autant que possible l'introduction de logique dans les templates.

En contrepartie, Django vous offre un système de gestion et d'affichage de contenu qui reprend les bonnes pratiques évoquées dans les chapitres précédents : DRY, héritage, etc. Il propose une manière de gérer les templates qui est particulièrement souple. Du point de vue du développeur, elle facilite une organisation DRY et efficace ; du point de vue de l'intégrateur, elle reste très simple et standard.

Comme vous l'avez déjà constaté, les templates Django ne sont que des fichiers HTML. Tout ce qui se rapporte aux techniques d'intégration reste donc valable lorsque vous développez une application Django. Il est ainsi tout à fait possible d'utiliser un framework CSS de votre choix, de faire un site HTML 5 ou même de développer des applications mobiles sans devoir modifier la logique interne de Django.

La partie JavaScript et Ajax est très souple elle aussi, car Django a fait le choix de ne pas vous forcer à utiliser un framework JavaScript comme d'autres frameworks web. La liberté d'implémentation que cela vous laisse est donc totale. Malgré tout, ce n'est pas parce que Django ne fait pas le choix d'un framework JavaScript qu'il ne vous offre pas un nombre conséquent de raccourcis vous simplifiant la vie au quotidien.

Organisation des templates

La première partie de la conception d'un site Internet consiste souvent à réaliser un gabarit général dans lequel vous définissez les grandes lignes de vos pages : en-tête, menu, pied de page. Django crée un fichier `base.html` à la racine de votre dossier `templates`. Ce fichier est ensuite utilisé dans l'ensemble de votre projet par les autres templates, qui en héritent.

Cette manière de faire est particulièrement souple et convient à la plupart des utilisations. Elle vous permet de découpler intelligemment vos templates et de réutiliser à profit ceux qui pourraient être redondants. Ainsi, vous gardez un esprit DRY jusque dans vos templates.

Le fichier base.html

Une bonne introduction à cette technique est la création d'un fichier `base.html` à la racine de vos templates. Il va se comporter comme un gabarit qui recevra par la suite

des fragments de templates. Vous définissez ainsi une apparence générale pour votre site Internet.

Commencez par créer ce fichier à la racine de votre dossier `template` et définissez-y la structure principale de votre document :

```
<html>
  <head>
    <title>Agenda partagé</title>
  </head>
  <body>
    <div id="header">
    </div>
    <div id="menu">
    </div>
    <div id="content">
    </div>
    <div id="footer">
    </div>
  </body>
</html>
```

L'important dans les templates Django, c'est que vous éditez toujours du HTML. Vous n'avez pas à apprendre une syntaxe étrange ou modifier votre code lorsque votre template change (et inversement). La logique existe pourtant bel et bien, mais elle reste confinée à des balises à la fois simples et clairement identifiables.

Une fois que vous avez défini votre template de base, vous n'avez plus qu'à prévoir les emplacements que vous pensez devoir modifier pour y ajouter du contenu issu de votre application. Pour commencer, il suffit de définir un emplacement pour le contenu principal de votre site :

```
<html>
  <head>
    <title>Agenda partagé</title>
  </head>
  <body>
    <div id="header">
    </div>
    <div id="menu">
    </div>
    <div id="content">
    {% block content %}
    {% endblock content%}
    </div>
```

```
    <div id="footer">
    </div>lock
  </body>
</html>
```

Les deux lignes ajoutées indiquent à Django de charger du contenu entre les instructions `{% content %}` et `{% endblock content %}`. Pour que du contenu soit chargé dans ce bloc, il faut qu'un template de votre application utilise le fichier `base.html`.

SYNTAXE Ouverture et fermeture de bloc

Quand vous créez de nouveaux blocs, la balise de fermeture ne contient pas nécessairement le nom du bloc. Ainsi, l'instruction `{% endblock %}` est tout aussi valide que `{% endblock monbloc %}`. Dans les exemples de ce livre, nous utiliserons toujours cette seconde syntaxe car elle permet de gagner en clarté et en simplicité, mais c'est à vous de choisir la syntaxe qui vous convient le mieux.

La directive extends

Prenons l'exemple du fichier `personal_calendar/event/liste.html`. Au début du fichier, la commande `{% extends "base.html" %}` demande à Django de charger le template `base.html` avant d'analyser `liste.html`.

Une fois cette ligne ajoutée, si vous tentez de visiter l'URL `http://localhost:8000/agenda/liste/`, vous remarquez que la page est blanche ! Le template `base.html` a totalement remplacé `liste.html`. Django a bien analysé ce template et a cherché des blocs de remplacement comme celui que vous avez défini auparavant `{% block content %}{% endblock content %}`. Par la suite, il a cherché, dans votre template `liste.html`, un bloc portant le même nom. N'en ayant pas trouvé, il vous a retourné le fichier `base.html`, sans contenu. Pour vous en assurer, vous pouvez insérer du contenu dans le bloc de votre fichier `base.html` et constater qu'il est bien affiché à l'écran :

```
{% block content %}

Bonjour depuis "base.html"

{% endblock content %}
```

En revanche, si vous définissez un bloc `content` au sein de votre fichier `liste.html`, son contenu sera affiché à la place de celui du même bloc dans le fichier `base.html` :

```
    {% block content %}Bonjour depuis liste.html{% endbloc content %}
```

Maintenant que vous saisissez la logique de la directive `extends`, vous pouvez insérer du « vrai contenu » :

```
{% extends "base.html" %}
{% block content %}
<ul>
{% for event in events %}
<li>
   <h3><a href="{{event.get_absolute_url}}">{{event.nom}}</a></h3>
      <p>le {{event.date}}</p>
      <p>à {{event.lieu}}</p>
{% endfor %}
</ul>
<p>
<a href="/agenda/create/">Créer un nouvel événement</a>
</p>
{% endblock content %}
```

Le contenu du template `liste.html` est alors chargé dans `base.html`, dans le bloc `content` que vous avez défini.

Cet exemple reste bien entendu relativement simple. Vous n'êtes pas obligé, par exemple, de définir tous les emplacements d'un coup. Vous pouvez déterminer plusieurs blocs dans votre fichier `base.html` qui seront ou non remplacés par le contenu de votre fichier `liste.html`. Supposons que vous souhaitiez modifier le titre de la page selon l'URL que vous visitez, vous définiriez un bloc `titre` dans votre template `base.html` de la manière suivante :

```
<head>
    <title>
    Agenda partagé
    {% block titre %}
    {% endblock titre %}
    </title>
</head>
```

Par la suite, dans votre template `liste.html`, vous remplirez ce nouveau bloc :

```
    {% block titre %}
    | Liste des événements
    {% endblock titre %}
```

Si l'exemple que vous venez de traiter ne prend en compte qu'un seul niveau d'héritage, c'est-à-dire que le fichier `base.html` n'est « rempli » que par le fichier `liste.html`, sachez qu'il est possible de créer de cette façon autant de niveaux d'héritage que vous le souhaitez. Une bonne pratique est de définir un template très géné-

raliste à la racine de votre projet, par exemple `base.html` comme vous venez de le faire. Ensuite, vous créez un template par application, par exemple le fichier `personal_calendar.html` où vous commencez à remplir le template de base avec du contenu générique. Vous pouvez aussi définir de nouveaux blocs à l'intérieur de ceux déjà écrits par `base.html`, afin d'affiner vos templates. Ensuite, vous écrivez seulement les parties spécifiques de chaque page.

Cette granularité rend très agréable le développement du design d'une application Django. Cependant, elle peut encore être étendue grâce à l'usage d'une seconde instruction : `include`.

La directive include

La directive `extends` est très pratique, mais elle ne répond pourtant pas à tous les cas d'utilisation. C'est vrai notamment quand vous souhaitez utiliser plusieurs fois un morceau de HTML dans vos templates, par exemple les formulaires que vous avez créés. Ces templates se répètent encore et encore à de nombreux endroits de votre application, avec très peu de différences.

La directive `include` règle justement ce type de problème. Vous pouvez définir des petits templates très simples et les charger dans les templates de votre choix. Par exemple, dans votre dossier `agenda/templates/personal_calendar/`, créez un dossier `blocks` qui recevra les fragments de HTML que vous souhaitez inclure dans les autres templates de votre application `personal_calendar`. Dans ce dossier, créez un fichier `delete_form.html` dans lequel vous pouvez écrire ce qui suit :

```
<form action="/agenda/{{event.id}}/delete/"
      method="POST">
  {% csrf_token %}
  <input type="submit" value="Supprimer" />
</form>
```

Supprimez ensuite toutes les occurrences de cette partie de formulaire. Par exemple, dans le template `details.html`, vous pouvez écrire :

```
<h1>{{event.nom}}</h1>
<p>{{event.description}}</p>
<p>le {{event.date}}</p>
<p>à {{event.lieu}}</p>
{% for participant in event.evenement_participant_set.all %}
{{participant.participant}} {{participant.get_status_display}}
<form action="/agenda/{{event.id}}/participant/{{participant.id}}/
delete/" method="POST">
  {% csrf_token %}
```

```
  <input type="submit" value="Supprimer" />
</form>
</p>
{% endfor %}
<form action="" method="POST">
{% csrf_token %}
{{form.as_p}}
<input type="submit" value="Créer" />
</form>

{% include "personal_calendar/blocks/delete_form.html" %}

<a href="/agenda/{{event.id}}/update/">Modifier</a>
```

Néanmoins, vous vous rendez sans doute compte que le formulaire de suppression des participants est très proche de celui que vous venez d'écrire. En réalité, la seule chose qui les différencie, c'est l'URL qui est utilisée pour le paramètre `action`.

Dans une architecture où l'on ne se répète pas, il peut être opportun d'indiquer comment retrouver l'URL de suppression d'une ressource dans les modèles. Par exemple, vous pouvez écrire dans votre fichier `models.py` la méthode `delete_url` pour la classe `Evenement` :

```
    def delete_url(self):
        return "/agenda/%s/delete/" % self.id
```

Pour la classe `Evenement_Participant`, vous pouvez faire de même :

```
    def delete_url(self):
        return "/agenda/%i/participant/%i/delete/" % (self.event.id,
                                                      self.participant.id)
```

Puis reprenez votre fichier `delete_form.html` de la façon suivante :

```
<form action="{{ delete_url }}"
      method="POST">
  {% csrf_token %}
  <input type="submit" value="Supprimer" />
</form>
```

Maintenant, votre template prend un argument `delete_url` qu'il va falloir lui servir. Vous verrez dans la suite qu'en utilisant des fragments de code que vous écrivez vous-même (les template tags), vous pouvez résoudre ce type de problématique. Pour l'heure, voyons comment répondre à votre besoin uniquement avec les pièces fondamentales de Django.

Actuellement, vous avez un attribut sur chaque participant et sur chaque événement, qui correspond à son URL de suppression. Vous avez également un formulaire qui prend cet argument. Voici comment le template tag Django `{% with %}` va faire coïncider ces deux informations :

```
{% with delete_url= participant.delete_url %}
{% include "personal_calendar/blocks/delete_form.html" %}
{% endwith %}
```

Entre `{% with %}` et `{% endwith %}`, `delete_url` vaut `participant.delete_url`. Votre formulaire inclus peut donc fonctionner.

De la même manière, vous pouvez écrire :

```
{% with delete_url=event.delete_url %}
{% include "personal_calendar/blocks/delete_form.html" %}
{% endwith %}
```

Cette fois, entre `{% with %}` et `{% endwith %}`, `delete_url` vaut `event.delete_url`.

Votre template `details.html` est maintenant le suivant :

```
<h1>{{event.nom}}</h1>
<p>{{event.description}}</p>
<p>le {{event.date}}</p>
<p>à {{event.lieu}}</p>
{% for participant in event.evenement_participant_set.all %}
{{participant.participant}} {{participant.get_status_display}}
{{participant.delete_url}}
{% with delete_url=participant.delete_url %}
{% include "personal_calendar/blocks/delete_form.html" %}
{% endwith %}
</p>
{% endfor %}
<form action="" method="POST">
{% csrf_token %}
{{form.as_p}}
<input type="submit" value="Créer" />
</form>

{% with delete_url=event.delete_url %}
{% include "personal_calendar/blocks/delete_form.html" %}
{% endwith %}
<a href="/agenda/{{event.id}}/update/">Modifier</a>
```

Même s'il existe des techniques plus puissantes pour inclure des fragments de code au sein de vos templates, ne négligez pas l'usage des inclusions et de `with`. Cette façon de

faire couvre bon nombre des problèmes auxquels vous serez confronté. De plus, elle est plus simple à mettre en œuvre qu'un template tag et plus facile à corriger.

Rev 17

Un agenda fonctionnel avant d'y ajouter CSS et JavaScript.

Les styles CSS et les images

Les directives `extends` et `include` vous donnent une totale liberté quant à la manière de représenter ou de découper vos templates. Très rapidement toutefois, vous allez avoir besoin de définir des styles CSS pour mettre en page votre contenu. Le parti pris de Django est de ne pas s'occuper de cette partie, tout du moins en production. Des serveurs web, très performants pour servir des images statiques, sont disponibles sur le marché (Apache, LightHttpd…).

Pourtant, Django part du principe que, pendant le développement, vous n'aurez pas sous la main un serveur Apache correctement configuré pour servir les images et autres fichiers statiques. Ce n'est pas forcément vrai ; vous pouvez tout à fait configurer une petite instance de LightHttpd, par exemple, pour servir vos images. Cela aura le désavantage de vous forcer à en modifier la configuration dès que vous changerez de projet, mais c'est tout à fait correct et fonctionnel.

Servir les fichiers statiques durant le développement

Django met à disposition une application pour servir les fichiers statiques pendant le développement de votre application : `django.contrib.staticfiles` ; son fonctionnement est assez proche de celui des templates. En effet, sans configuration autre que l'ajout de `django.contrib.staticfiles` dans la liste des `INSTALLED_APPS`, Django va chercher les fichiers statiques dans un dossier `static` de vos applications. En toute logique, vous pouvez donc créer le dossier `/agenda/personal_calendar/static/`. Tous les fichiers statiques que vous y placerez seront alors disponibles dans vos templates grâce à l'appel de variable suivant :

```
{{ STATIC_URL }}
```

DÉFINITION Context processors

Le contexte est un dictionnaire que reçoit un template depuis la vue et qui sert à compiler un fichier statique, généralement du HTML mais pas seulement. Ce contexte contient le dictionnaire fourni par la vue mais également, s'il est de type `RequestContext`, un ensemble de données issues des *context processors* du projet. Ces derniers sont des fonctions qui prennent en argument l'objet `Request` et qui enrichissent le `context` avec des données supplémentaires. C'est l'une de ces fonctions qui rend disponible `{{STATIC_URL}}` ou `{{request.user}}` par exemple.

Bien entendu, `STATIC_URL` est la variable que vous avez définie dans le `settings.py` de votre projet. Avec un peu plus de configuration, vous pouvez aussi rendre ces dossiers disponibles en production. Néanmoins, une solution utilisant un serveur spécialement dédié à cette tâche reste plus performante. Pour styler vos templates, vous pouvez créer un dossier `css` dans lequel vous écrirez votre feuille de style, par exemple `global.css`, que vous appellerez dans vos templates, idéalement dans `base.html` :

```
<link rel="stylesheet" type="text/css" href="{{STATIC_URL}}css/
global.css">
```

Rev 8

Vous trouverez dans cette révision un fichier CSS complet qui donnera à votre application un style plus attrayant. Nous avons également réorganisé les templates en utilisant tout ce que vous avez appris au sein de ce chapitre. Vous pouvez vous en inspirer pour définir l'aspect de votre propre application d'agenda partagé.
Vous pouvez également, à titre d'exercice, retravailler l'apparence de votre tracker.

Inclure un framework CSS dans Django

Il est très courant aujourd'hui d'utiliser un framework CSS pour simplifier le développement graphique d'une application. Plusieurs d'entre eux emploient ou dérivent du concept de la grille 960, qui découpe la page en colonnes et espaces égaux totalisant 960 pixels. Elle est utilisée à toutes les étapes de la conception visuelle de l'application, tout d'abord sur un outil de travail d'image rastérisée ou vectorielle.

L'intégrateur qui utilise un framework 960 n'a plus qu'à placer ses éléments à l'aide de cette grille. Le framework lui offre de nombreux raccourcis permettant, par exemple, d'afficher la fameuse grille dans son template. Aujourd'hui, on trouve même des frameworks offrant des grilles « liquides », qui se redimensionnent en fonction de la taille de l'écran de l'utilisateur.

Outil Framework Twitter Bootstrap

Ce framework est basé sur le concept de grille 960 et propose une liste de fichiers CSS utiles et bien conçus. Comme son nom l'indique, il vous aide à démarrer un projet facilement et proprement. Il est ensuite possible de modifier tout ou partie de l'apparence, car Twitter Bootstrap reste basé sur LESS (http://lesscss.org/), framework CSS complet amenant de vraies nouveautés au sein du langage comme les imbrications, les variables…

- http://twitter.github.com/bootstrap/

La volonté de Django de rester parfaitement transparent dans la conception des pages HTML, de limiter au maximum la logique au sein des templates et de servir simplement vos fichiers statiques, vous permet d'utiliser ce type de framework CSS le plus simplement du monde. Placez les fichiers CSS dont vous avez besoin dans le dossier `static` de votre application et commencez à vous en servir. C'est aussi simple que cela.

Intégrer du JavaScript

Lorsque les premiers frameworks vraiment populaires sont apparus et que la technologie Ajax a vraiment rencontré son public, certains frameworks ont voulu simplifier le travail des développeurs en intégrant directement dans leur code l'usage d'un framework JavaScript. Ce ne fut pas le cas de Django qui désirait rester le plus agnostique possible. Cela lui fut longtemps reproché et apparaissait régulièrement dans la colonne des « moins » sur les tableaux de comparaison. Mais aujourd'hui que les frameworks JavaScript se sont démocratisés, tout le monde ou presque considère cela comme une force. Django utilise néanmoins, en particulier dans l'interface d'administration, le framework JavaScript jQuery, qui est aujourd'hui certainement le plus largement employé. Toutefois, cela n'a rien à voir avec une quelconque intégration : il est utilisé principalement pour créer les widgets de choix de dates et d'autres choses de ce type dans l'`admin`.

Reprenons notre application et ajoutons-lui une couche de JavaScript qui simplifiera son utilisation. En premier lieu, plutôt que de laisser les utilisateurs entrer les dates des événements à la main, nous allons afficher un calendrier JavaScript dans notre formulaire. Ensuite, nous rendrons la liste d'événements plus dynamique grâce à la technique de « pliage/dépliage ». Enfin, nous utiliserons des appels Ajax pour permettre l'édition en ligne sans qu'il soit nécessaire de recharger la page.

Calendrier JavaScript

Actuellement, un utilisateur souhaitant ajouter ou modifier la date d'un événement doit saisir lui-même l'information, avec les éventuelles erreurs de frappe et de validation que cela suppose. Pour lui simplifier l'édition du formulaire, nous allons ajouter un calendrier.

Le champ de saisie de date sera toujours affiché et la saisie manuelle toujours disponible. Quand vous développez ce type d'amélioration, demandez-vous toujours si vous ne supprimez pas une fonctionnalité au profit d'une autre. Il n'est rien de plus frustrant que de devoir se battre contre une fonctionnalité prétendument plus simple mais qui, finalement, ne fait que compliquer les choses. Proposer un choix reste donc bien souvent la meilleurs attitude à adopter.

Pour insérer la bibliothèque jQuery, créez un sous dossier `js` dans votre dossier `static` et téléchargez-y un exemplaire du fichier `jquery.js` depuis le site Internet de jQuery. Vous pourrez ensuite y faire référence dans la balise `head` de vos templates, comme ceci :

```
<script type="javascript" src="{{STATIC_URL}}/js/jquery.js"></script>
```

Comme vous allez utiliser jQuery sur pratiquement toutes les pages de vos templates, insérez cette ligne directement dans le template `base.html` de votre projet. Pour vérifier que jQuery est bien chargé, écrivez les lignes suivantes entre les balises `<head>` et `</head>` :

```
<!-- Chargez jquery -->
<script type="text/javascript" src="{{STATIC_URL}}/js/jquery.js">
</script>
<!-- attendez le chargement complet de la page, puis affichez un
message -->

<script type="text/javascript">
    $('document').ready(function(){
        alert($('h1').first().text())
    });
</script>
```

Maintenant que vous avez validé que jQuery est bien activé, vous pouvez commencer le développement réel de votre fonctionnalité JavaScript. Pour ajouter une sélection de date à vos formulaires, le plus simple est d'utiliser des bibliothèques externes comme l'excellent widget `datepicker` dans le module `ui` de jQuery. Il se présente sous la forme de trois fichiers : `jquery.ui.core.js`, `jquery.ui.datepicker.js` et `jquery.ui.datepicker-fr.js`.

Téléchargez simplement ces trois fichiers dans votre dossier `static/js/`, puis chargez-les dans l'en-tête de votre fichier `base.html` :

```
<script type="text/javascript"
 src="{{STATIC_URL}}/js/jquery.ui.core.js"></script>
 <script type="text/javascript"
 src="{{STATIC_URL}}/js/jquery.ui.datepicker.js"></script>
 <script type="text/javascript"
 src="{{STATIC_URL}}/js/jquery.ui.datepicker-fr.js"></script>
```

Il vous faut ensuite indiquer les champs de formulaire qui vont utiliser ce widget JavaScript. Sur ce point, Django vous simplifie beaucoup la tâche. En effet, pour chaque champ `date` dans vos formulaires, aussi bien pour la création d'un événement que pour sa modification, il ajoute le code HTML suivant :

```
<p>
  <label for="id_date">
    Date:
  </label>
  <input type="text" name="date" value="" id="id_date" />
</p>
```

Comme vous pouvez le remarquer, le champ possède l'identifiant `id_date`. Il vous suffit donc d'appeler la fonction `datepicker` sur tous les champs portant cet identifiant :

```
    <script type="text/javascript">
    $('document').ready(function(){
        $('#id_date').datepicker()
    });
    </script>
```

Une seule ligne de code est donc nécessaire pour utiliser ce widget sur tous les champs dates dans tous vos formulaires. Django et jQuery forment un couple extrêmement puissant !

Rev 19

Ajout de jQuery et `datepicker`.
Pour vous simplifier la vie, vous trouverez dans cette révision une installation fonctionnelle du calendrier jQuery et des fichiers requis.

Pliage et dépliage

Cacher à l'utilisateur une partie du contenu d'une page assure une organisation claire, en ne lui affichant que les informations dont il a réellement besoin. Dans le cas de notre application, l'usage d'une telle technique prend tout son sens sur le template `liste.html`. Vous pouvez vous servir du système de « pliage/dépliage » pour afficher ou non les éléments optionnels. Ce template, tel qu'il est actuellement écrit pourrait charger une collection d'événements en utilisant un bloc template similaire à `personal_calendar/event/details.html` et affichant les informations d'un événement :

```
<li>
<h3><a href="{{event.get_absolute_url}}">{{event.nom}}</a></h3>
<p>{{event.description}}</p>
<p>le {{event.date}}</p>
<p>à {{event.lieu}}</p>
{% for participant in event.evenement_participant_set.all %}
```

```
<div class="well">
{{participant.participant}}: {{participant.get_status_display}}

</p>
</div>
{% endfor %}
<a href="/agenda/{{event.id}}/update/">Modifier</a>
```

Dans un template comme celui-ci, c'est sur la ligne en exergue qu'il faut « déclencher » le pliage/dépliage de l'événement. Attention toutefois, ce lien est, sur la page, la seule porte d'entrée vers la page de l'événement. Même si nous allons très rapidement faire en sorte que la page `liste` soit la page principale sur laquelle toute (ou presque) l'application repose, il est très important que vos ressources gardent leurs URL.

L'URL, identifiant unique

Avec l'émergence des techniques Ajax, le Web moderne ressemble de plus en plus à un ensemble de clients lourds, et de nombreuses applications web remplacent des applications classiques (certains webmails ou l'agenda partagé que vous êtes en train de réaliser, par exemple). Cependant, ce qui fait la force d'une application web, c'est le partage des données. Aujourd'hui, en envoyant un simple lien à votre correspondant, vous lui transmettez énormément d'informations : les horaires de la prochaine séance du film que vous souhaitez voir ensemble, les critiques d'un restaurant... Tout ceci est possible parce que chaque ressource est désignée par une URL unique. De la même manière, vos utilisateurs vont se référer à un événement grâce à son identifiant unique : son URL. En conséquence, quand vous écrivez vos applications, posez-vous toujours cette question : « Ma ressource est-elle identifiable via une URL unique ? »

Modifiez donc cette ligne comme suit :

```
<h3>
    <a href="" class="titre">
      {{event.nom}}
    </a>
    <a href="{{event.get_absolute_url}}">
      (details)
    </a>
</h3>
```

Vous venez de donner la classe `titre` à un lien vide et déplacer le lien qui pointe vers la ressource. De cette façon, vous pouvez ajouter un déclencheur JavaScript sur le lien vide et laisser le second fonctionner normalement. Ce travail de préparation étant terminé, il ne vous reste plus qu'à écrire la partie JavaScript.

Au chargement de la page, les détails d'un événement doivent être cachés ; seul le titre doit être affiché. Ensuite, un clic sur le titre doit faire apparaître le détail de l'événement. Cacher les éléments via JavaScript, et non directement dans le fichier CSS, rend le site compatible avec les navigateurs n'utilisant pas JavaScript : dans ces derniers, les éléments ne seront pas pliés/dépliés, mais resteront affichés en permanence.

Si vous observez la structure du code HTML, il faut remonter l'arbre DOM d'un niveau pour atteindre le tag `h3`, puis remonter encore d'un niveau pour atteindre le `li` et enfin atteindre les enfants portant la classe `event`. En utilisant la syntaxe JavaScript, cela se traduit comme suit :

```
$(this).parent().parent().children('.event')
```

GLOSSAIRE L'arbre DOM

DOM signifie *Document Object Model*. L'arbre DOM est une vue hiérarchique d'un document, en l'occurrence HTML. Le concept est simple : la page est une succession de parents contenant des enfants qui eux-mêmes contiennent des enfants, et ainsi de suite. JavaScript permet de parcourir et de manipuler cet arbre DOM.

Vous pouvez donc utiliser la fonction `toggle()` qui va cacher les éléments sélectionnés s'ils sont visibles et les montrer dans le cas contraire. Il vous faut également cacher les éléments de type `ul` contenus dans les `div` de classe `event`. Le code résultant est le suivant :

```
<script type="text/javascript">
$('document').ready(function(){
$('.event').hide()
    $('#id_date').datepicker()
    $('.titre').click(function(e){
        $(this).parent().parent().children('.event').toggle()
    e.preventDefault()
    });
});
</script>
```

Comme vous le constatez, il est extrêmement simple d'intégrer du code JavaScript au sein de vos templates, sans qu'il soit nécessaire de toucher au code Django proprement dit. Cela vous donne une vraie liberté quant aux choix techniques qui s'offrent à vous. Ici, nous avons choisi de vous montrer le fonctionnement de Django et de JavaScript avec la bibliothèque jQery, mais vous êtes libre d'utiliser le framework de votre choix.

Par ailleurs, cela offre aussi la possibilité de séparer les tâches. Si vous avez sous la main un intégrateur féru de JavaScript, il n'est pas indispensable qu'il sache utiliser Django. Contentez-vous de lui fournir les données dont il a besoin, il se chargera de

les mettre en pages et d'intégrer les frameworks JavaScript et CSS de son choix sans plus de difficultés.

Les requêtes Ajax

> **GLOSSAIRE Ajax**
>
> Ajax est l'acronyme de *Asyncronous JavaScript And XML*. C'est une technologie qui sert à modifier le document, c'est-à-dire la page HTML, sans avoir besoin de rafraîchir l'ensemble de la page. Son principe est simple : une requête est effectuée par la page au serveur, qui renvoie un fragment de donnée, souvent au format JSON, utilisé par JavaScript pour modifier la page.

Dans le cas un peu particulier des requêtes Ajax, on serait enclin à penser que cette fois, il va falloir créer de nouvelles vues afin de fournir les données demandées. En réalité, Django vous simplifie, une nouvelle fois, grandement la tâche. L'objet `request`, fourni en paramètre de toutes vos vues, contient un paramètre déterminant la nature de la requête. Ainsi, il est possible d'utiliser une seule vue pour renvoyer un contenu de format différent suivant que cette requête est effectuée via Ajax ou non.

Dans l'exemple qui suit, vous allez offrir à l'utilisateur la possibilité d'ajouter ou de supprimer des participants directement depuis la page `details`, et ce, sans recharger la page. Le fait d'ajouter ou de supprimer des participants est une fonctionnalité que vous avez déjà écrite ; il est donc inutile d'y revenir. En revanche, vous avez besoin d'écrire l'appel Ajax dans votre page, à l'aide de jQuery. La première chose à faire, c'est de se donner un peu d'espace pour travailler sur notre formulaire. En utilisant la logique des inclusions, créez un nouveau fichier :

personal_calendar/blocks/participant_form.html

```
<form action="" method="POST">
{% csrf_token %}
{{form.as_p}}
<input type="submit" value="Créer" />
</form>
```

Puis incluez-le dans votre fichier `details.html` :

```
{% include "personal_calendar/blocks/participant_form.html" %}
```

Donnez un identifiant unique à votre formulaire, de façon à le manipuler facilement avec jQuery :

```
<form action="" method="POST" id="participant_form">
```

Ensuite, attachez un événement JavaScript sur votre formulaire :

```
$("#participant_form").submit(function(e){
            $.ajax(
                {
                    type:"POST",
                    data:$(this).serialize(),
                    url:"",
                    success: function(data){
                        console.log(data)
                    },
                }
            );
            e.preventDefault();
        });
```

Pour le moment, on ne fait qu'afficher le retour du formulaire afin de valider que le fonctionnement de JavaScript est bien le bon. Si vous testez ce code, vous ne verrez ni les éventuels messages d'erreurs, ni les nouveaux participants s'ajouter.

Maintenant, intéressons-nous à la partie Django et tentons de répondre différemment suivant que la requête est effectuée ou non en Ajax. Dans votre vue `details`, testez si la requête est de type Ajax, en écrivant par exemple :

```
    print request.is_ajax()
```

Vous allez remarquer que, lorsque vous rechargez la page, le terminal affiche `False` et, quand vous soumettez le formulaire, le terminal vous renvoie `True`. Vous pouvez donc maintenant renvoyer un contenu spécifique. Dans le cas d'une erreur, il est plus juste de renvoyer le formulaire complet. Ajoutez `render_to_response` à la liste des `import` :

```
from django.shortcuts import render, HttpResponseRedirect,
render_to_response
```

Puis, dans la fonction `details`, vous pouvez écrire :

```
if request.is_ajax():
        return render_to_response(
            'personal_calendar/blocks/participant_form.html',
            {'event':event,
             'form': form})
```

Dans le cas où le formulaire est valide, il faut renvoyer un JSON contenant l'objet `participant` qui vient d'être créé. Il faut aussi pouvoir renvoyer le formulaire de sup-

pression du participant ; en effet, il ne faudrait pas recopier ce template dans votre JavaScript. Nous allons donc écrire un dictionnaire contenant les informations dont votre template a besoin. Pour cela, vous devez importer `json`, pour renvoyer une réponse dans le bon format, `render_to_string`, qui servira à renvoyer uniquement le formulaire compilé, et `RequestContext`, car vous allez avoir besoin de récupérer `{% csrf_token %}` :

```
import json
from django.template.loader import render_to_string
from django.template import RequestContext
```

Le code de votre fonction est maintenant le suivant :

```
def details(request, id):
    event = Evenement.objects.get(pk = id)
    if request.method == "POST":
        form = Evenement_ParticipantForm(request.POST)
        if form.is_valid():
            form.save()
            if request.is_ajax():
                delete_form = render_to_string(
                    "personal_calendar/blocks/delete_form.html",
                    {'delete_url': form.instance.delete_url(),
                     },
                    RequestContext(request)
                    )
                data = {'participant':
                         form.instance.participant.username,
                        'get_status_display':
                         form.instance.get_status_display(),
                        'delete_form': delete_form}
                return HttpResponse(
                    json.dumps(data),
                    mimetype="application/json"
                    )
            return HttpResponseRedirect('/agenda/%s/details/' % id)
    else:
        form = Evenement_ParticipantForm(initial = {'evenement':event})
        participants = [user.pk for user in event.participants.all()]
        form.fields['participant'].queryset=User.objects.exclude(
            pk__in = participants)
        form.fields['evenement'].widget = HiddenInput()
    if request.is_ajax():
        return render_to_response(
            'personal_calendar/blocks/participant_form.html',
            {'event':event,
             'form': form})
```

```
    return render(request,'personal_calendar/event/details.html',{
            'event':event,
            'form':form})
```

Il ne reste plus qu'à revenir sur votre code JavaScript pour traiter les données que Django vous renvoie. Comme on sait que l'on va recevoir du JSON ou du HTML en fonction de la validation ou non du formulaire par Django, on peut aisément utiliser un branchement conditionnel :

```
if(typeof(data) == "string"){
  # Le formulaire contient des erreurs.
  }else{
  # Le formulaire a été correctement enregistré.
}
```

Il faut encore ajouter les données au bon endroit. Pour plus de simplicité, nous créons un nouveau `div` autour de notre collection de participants, ce qui permet d'écrire le code suivant, parfaitement fonctionnel :

```
$("#participant_form").submit(function(e){
    $.ajax(
        {
            type:"POST",
            data:$(this).serialize(),
                url:"",
                success: function(data){
                    if(typeof(data) == "string"){
                        $("#participant_form").html(data);
                    }else{
                        $("#participants").append(
                            "<div class='well'>"
                           + data.participant + ": "
                           + data.get_status_display + ""
                           + data.delete_form +
                            "</div>"
                        )
                    }
                },
            }
        );
        e.preventDefault();
    });
```

Pour supprimer des éléments via Ajax, c'est encore plus simple. Ajoutez une classe `delete` à vos formulaires :

```
<form action="{{ delete_url }}"
      method="POST" class="delete">
  {% csrf_token %}
  <input type="submit" value="Supprimer" />
</form>
```

puis le code JavaScript suivant :

```
        $(".delete").submit(function(e){
            form = $(this)
            $.ajax({
                type:"POST",
                data:$(this).serialize(),
                url:$(this).attr("action"),
                success: function(data){
                    if(data == "OK"){
                        form.parent().remove()
                    }
                }});
            e.preventDefault()
            })
```

Modifiez enfin la fonction `delete_participants` en ajoutant les lignes suivantes :

```
        if request.is_ajax():
            return HttpResponse("OK")
```

Vous pouvez désormais ajouter et supprimer les participants directement depuis Ajax. Une petite chose encore : comme vous le constatez, les participants créés via Ajax ne sont pas supprimés en utilisant Ajax. Il y a donc un rafraîchissement de page. Cela peut être corrigé facilement en utilisant la fonction `live`, qui va attacher l'événement que vous avez écrit sur les éléments que vous ajoutez via Ajax. Pour l'utiliser, changez simplement :

```
$(".delete").submit(function(e){
```

par :

```
$(".delete").live("submit", function(e){
```

Rev 20

Une interface Ajax.

Conclusion

Au cours de ce chapitre, nous nous sommes attachés à donner une vision claire de tous les éléments qui concernent l'affichage des informations. À propos des templates, vous avez appris à utiliser à bon escient les directives `extends` et `include` pour mieux découpler votre application. Vous savez maintenant gérer les médias statiques comme les feuilles de style, les images et le JavaScript. Enfin, à travers les deux derniers exemples, vous avez pu mettre en place simplement des réponses différentes selon que la demande émane du navigateur ou d'une requête Ajax, grâce à l'instruction `is_ajax`.

Point important à ne pas perdre de vue : Django permet facilement de laisser la création de templates, de CSS et de code JavaScript à un tiers, qu'il soit intégrateur ou développeur *front-end* (spécialiste du développement d'interface client). C'est l'une de ses forces dans l'univers professionnel.

7

Les vues génériques pour les fonctions d'agenda

Depuis que vous avez commencé l'application d'agenda partagé, vous avez écrit chacune de vos fonctions dans les vues. C'est souvent la manière la plus simple et la plus rapide pour ébaucher une application. Cependant, à mesure que votre connaissance de Django va évoluer, vous allez utiliser des mécanismes beaucoup plus puissants : les vues génériques.

Les vues génériques depuis Django 1.3

Les vues génériques existent dans Django depuis longtemps. C'est seulement depuis la version 1.3 qu'elles sont devenues incontournables dans les applications Django. Autrefois, il s'agissait de fonctions génériques pour remplir rapidement des tâches simples. Néanmoins, dès que vos besoins gagnaient en complexité, il était nécessaire de les abandonner ou de les détourner.

Aujourd'hui, les vues génériques ne sont plus de simples fonctions, mais des classes. Leurs objectifs sont inchangés : vous simplifier la vie et accélérer le développement de vos applications. La vraie grande nouveauté réside dans le très puissant système d'héritage mis en place. Ce principe de classe vous permet d'écrire moins de code pour des fonctionnalités équivalentes et, surtout, de jouer beaucoup sur la « réutilisabilité ». Le fait qu'il s'agisse de classe et non plus de fonction vous donne une très grande souplesse dans l'élaboration de vos fonctionnalités. Vous pouvez les

employer telles quelles, les modifier, en hériter pour créer les vôtres. Les vues génériques sont désormais une nouvelle façon de développer une application.

Les bases des vues génériques

Les vues génériques peuvent être utilisées à plusieurs niveaux. Au niveau d'abstraction le plus haut, elles servent, sans configuration particulière, à implémenter toutes les fonctionnalités d'une application CRUD : dans notre cas, afficher une liste d'événements, et afficher, créer, modifier et supprimer un événement. L'origine de Django, issu de la presse Internet, fait qu'il existe également des vues génériques basées sur les dates, années, mois, semaines, jours…

Ces vues génériques sont à la fois très simples et très souples. Elles ont été écrites pour que vous puissiez les étendre, modifier leurs comportements, y ajouter des fonctionnalités mais, surtout, créer les vôtres. En effet, toutes les vues génériques de Django sont issues de classes de base, fondamentales pour créer vous-même des vues génériques correspondant à un besoin particulier. Bien entendu, dans le cas d'un besoin vraiment très particulier, on ne peut plus vraiment parler de vue générique… Mais c'est une nouvelle façon de développer des applications qui s'offre à vous.

Pour le moment, vous allez employer les vues génériques de la façon la plus simple qui soit. À mesure que votre application va gagner en complexité, vous découvrirez comment les étendre, comment créer vos propres vues génériques et comment écrire plus rapidement, avec moins de code, les fonctionnalités dont vous aurez besoin.

Afficher une collection d'événements

Vous avez déjà abordé les vues génériques dans leur forme la plus simple avec `TemplateView`, qui relie simplement une URL et un template. Ici, nous allons découvrir une vue générique un peu plus utile : `ListView`. Comme son nom le laisse supposer, c'est une classe qui affiche une collection d'objets ; elle va donc remplacer la vue `liste`.

Si vous regardez de nouveau votre code, vous constaterez que la vue `liste` que vous avez écrite est déjà très sommaire. Vous récupérez tous les événements et vous les affichez dans un template. Au premier abord, employer une vue générique pour cela n'apporte rien de particulier. Commencez par écrire un code qui va retourner une erreur, de manière à constater comment la classe `ListView` est pour ainsi dire autodocumentée. En effet, en essayant de l'utiliser sans argument, Django va vous aiguiller et vous permettre de corriger votre appel.

Tout d'abord, nous allons créer une nouvelle URL, qui va se comporter exactement de la même manière que la fonction `liste`. Cette URL va évidemment faire doublon avec la vue `liste` que vous avez déjà écrite, mais cela vous aidera à mieux saisir comment se comportent les vues génériques dans leur forme la plus simple. Dans les URL de `personal_calendar`, importez la classe `ListView`, ainsi que le modèle dont vous avez besoin, en l'occurrence `Evenement`, puis créez votre nouvelle URL. Votre fichier `urls.py` devrait ressembler à ce qui suit :

```
from django.conf.urls.defaults import patterns, include, url
from views import create, delete, details, update, liste, delete_participant
from django.views.generic.list import ListView
from models import Evenement

urlpatterns = patterns('',
    url(r'^create/$', create),
    url(r'^(\d+)/details/$', details),
    url(r'^liste/$', liste),
    url(r'^listes/$', ListView.as_view()),
    url(r'^(\d+)/delete/$', delete),
    url(r'^(\d+)/participant/(\d+)/delete/$', delete_participant),
    url(r'^(\d+)/update/$', update),
    )
```

Tentez maintenant de charger cette page et remarquez le message d'erreur (voir figure 7-1).

Figure 7–1
Erreur retournée lors de l'utilisation de ListView sans argument

Cette fonction attend donc en argument une queryset ou un modèle. C'est une très bonne chose, car vous vous en servirez pour présenter soit tous les objets (en lui fournissant un modèle), soit une liste filtrée (avec une queryset).

Pour présenter la liste des événements, il suffit donc de reprendre l'URL et de la modifier comme suit :

```
url(r'^listes/$',ListView.as_view(model = Event)),
```

Comme vous pouvez le constater, on utilise simplement la fonction `as_view`, comme pour la vue générique `TemplateView`, et on lui donne en argument le modèle dont on a besoin. Cependant, une nouvelle fois, ce code ne fonctionnera pas. L'idée est encore de se laisser guider par Django dans la découverte du fonctionnement de cet appel. Essayons de charger la page `localhost:8000/event/listes`. Django lève l'erreur `TemplateDoesNotExist`, car il attend le template `personal_calendar/evenement_list.html`.

La première chose remarquable est que, sans configuration particulière, Django vous propose des noms de templates prédéfinis, en utilisant le nom du modèle, un tiret de soulignement suivi de `list`. Comme souvent avec Django, ceci n'est qu'une configuration par défaut. Vous pouvez tout à fait lui fournir un nom de template qui convient mieux à vos besoins en utilisant l'argument nommé `template_name`. Toutefois, dans les exemples qui vont suivre, nous suivrons la configuration par défaut de Django.

La gestion des templates par Django

Peut-être vous êtes-vous déjà posé ces questions : comment Django gère-t-il les templates ? Comment est-il en mesure de retrouver l'un ou l'autre des templates que vous utilisez, dans les vues ou directement dans vos URL ? La logique est la suivante.
Tout d'abord, Django recherche le template dans le dossier `templates` défini dans `settings.py`. S'il ne trouve pas celui que vous lui avez demandé, il va alors chercher un dossier `templates` dans chacune de vos applications, selon l'ordre dans lequel vous les avez définies dans `settings.py`. Dès que le template demandé est trouvé, Django le compile et cesse de chercher.
Tout d'abord, cette logique vous autorise à écrire vos templates au sein même de l'application, la rendant particulièrement portable. De plus, un utilisateur de votre application va pouvoir écrire ses propres templates dans le dossier `templates` qu'il aura défini dans `settings.py` pour que Django utilise les siens plutôt que les vôtres. Enfin, cela vous laisse la possibilité, pour chacun de vos projets, de créer une application `theme` que vous définirez en tout premier dans la liste de vos `INSTALLED_APPS` et dans laquelle vous pourrez créer tous les templates spécifiques de votre application.

Afin de rendre l'application encore plus portable, nous allons enregistrer nos templates non plus dans le dossier `templates/personal_calendar/`, mais dans un nouveau dossier `templates` au sein de l'application : `agenda/personal_calendar/templates/`. Le

dossier ainsi créé se retrouvera au même niveau que le dossier `static` qui héberge les fichiers CSS, JavaScript et images.

Il ne vous reste plus qu'à écrire votre template. Là encore, la configuration par défaut transmet la liste des objets dans la variable `object_list`. Vous pouvez donc écrire :

```
{% for event in object_list %}
  {{event.nom}}
{% endfor %}
```

Bien entendu, il est tout à fait envisageable de réemployer ici tout ce que vous avez déjà vu sur les templates. La fonction `as_view()` rend disponible tout ce dont vous avez besoin, le `Context` par exemple.

Une fois que vous avez testé le bon fonctionnement du template, réécrivez-le en utilisant ce que vous avez déjà fait plus haut :

```
{% extends "base.html" %}

{% block titre %}
    | Liste des évenements
{% endblock titre %}

{% block content %}
<ul>
{% for event in object_list %}
{% include "personal_calendar/blocks/event_detail.html" %}
{% endfor %}
</ul>
<p>
<a href="/agenda/create/">Créer un nouvel évenement</a>
</p>
{% endblock content %}
```

La pagination

La fonction que vous aviez écrite pour afficher la liste des événements n'était pas vraiment complexe et vous pouvez trouver un peu inutile d'utiliser les vues génériques pour remplacer une vue aussi simple. À ce stade, c'est parfaitement exact. Cependant, pensez au code que vous devrez écrire si vous souhaitez implémenter une solution de pagination car, à l'usage, vous allez vous apercevoir qu'afficher une liste de tous les événements est indigeste – imaginez simplement à quoi ressemblera votre application quand il faudra présenter plus de 300 événements par exemple. Dans ce cas, il est plus simple d'afficher les 10 premiers, puis de fournir un lien pour les 10 suivants, et ainsi de suite.

Sans utiliser les outils fournis par Django, le code que vous devriez écrire est plutôt long. Par exemple, vous pourriez rechercher un paramètre `get` pour savoir quelle portion de la requête afficher et penser à gérer le cas où l'on atteint la fin de la liste, celui où le paramètre donné ne correspond pas à une partie de la liste, etc.

La pagination dans les vues

Heureusement, Django fournit une fonction qui pagine automatiquement et crée tout ce qui vous sera utile :

```
from django.core.paginator import Paginator, InvalidPage, EmptyPage
def liste(request):
    events = Evenement.objects.all()
    paginator = Paginator(events, 10)
    try:
        page = int(request.GET.get('page', '1'))
    except ValueError:
        page = 1
    try:
        events = paginator.page(page)
    except (EmptyPage, InvalidPage):
         events = paginator.page(paginator.num_pages)
    return render(request,
                  'personal_calendar/event/liste.html',
         {"events" :
         events})
```

En premier lieu, instanciez un objet de la classe `Paginator`.Vous devez lui fournir en argument une liste d'objets et un nombre qui lui servira à découper votre liste en « pages » de la longueur spécifiée. Essayez ensuite de récupérer la valeur de la clé `page` dans le dictionnaire `GET` de la requête. Si cette clé n'est pas définie, déterminez la valeur de `page` à 1. Puis tentez de charger la page demandée avec la fonction `paginator.page`. Dans le cas où elle n'existe pas ou est invalide, renvoyez la dernière page. Prenez garde cependant car, désormais, vous ne fournissez plus à votre template une liste d'objets `Evenement`, mais un objet `paginator`. Pour accéder à la liste des événements, vous devez donc écrire :

```
{% for event in events.object_list %}
    {{event.nom}}
{%endfor%}
```

Il vous reste à écrire dans votre template la pagination qui affiche, s'il y a lieu, la page précédente, la page suivante, la dernière page, le nombre de pages, etc.

```
        {% if events.has_previous %}
            <a href="?page={{ events.previous_page_number }}">précédent</a>
        {% endif %}
            Page {{ events.number }} sur {{ events.paginator.num_pages }}
        {% if events.has_next %}
            <a href="?page={{ events.next_page_number }}">suivant</a>
        {% endif %}
```

Pour bien se rendre compte à quel point les vues génériques rendent le travail plus aisé, voici comment paginer avec une vue générique. Modifiez simplement votre URL comme suit :

```
url(r'^listes/$',ListView.as_view(model = Evenement, paginate_by=10))
```

Le simple fait d'ajouter l'argument `paginate_by` rend disponible dans votre template un objet `paginator` qui représente la pagination et un objet `page_obj` qui représente la page courante. Il vous suffit d'utiliser ces deux objets dans vos templates comme vous l'avez vu plus haut :

```
<p>

        {% if page_obj.has_previous %}
            <a href="?page={{ page_obj.previous_page_number }}">précédent</a>
        {% endif %}
            Page {{ page_obj.number }} sur {{ page_obj.paginator.num_pages }}
        {% if page_obj.has_next %}
            <a href="?page={{ page_obj.next_page_number }}">suivant</a>
        {% endif %}
</p>
```

Ce bloc sera employé à chaque fois que vous aurez besoin de paginer vos résultats. Il est donc judicieux d'enregistrer ce petit bout de template dans le dossier `blocks` de vos templates et de l'appeler dès que nécessaire, par exemple :

```
{% include "personal_calendar/blocks/pagination.html" %}
```

En comparaison avec ce que vous avez déjà réalisé dans vos vues avec la même fonction, voyez la différence ! Si Django vous simplifie le travail pour des choses simples, c'est aussi pour vous permettre d'en faire plus. Imaginons maintenant que vous souhaitiez filtrer les événements. Par exemple, il peut sembler intéressant de n'afficher que ceux dont la date n'est pas passée. Là encore, c'est un paramètre nommé qui va faire tout le travail à votre place.

```
url(r'^listes/$',ListView.as_view(
                queryset=Evenement.objects.filter(
                    date__gte = datetime.datetime.now()
                    ),paginate_by=10)
                ),
```

Puisque vous avez commencé à utiliser les vues génériques, pourquoi ne pas continuer ? Vous pourriez filtrer les résultats par utilisateur, simplement en modifiant la queryset de votre classe, comme vous venez de le faire avec le filtre par date. Malheureusement, vous êtes maintenant un peu bloqué… L'objet `User` se trouve dans la requête et, pour accéder à cette dernière, vous avez besoin d'une vue. La solution qui vient tout de suite à l'esprit est de reprendre nos bonnes vieilles vues et de recommencer à coder spécifiquement chaque fonctionnalité. C'était sans doute vrai quand les vues génériques de Django n'étaient que des fonctions (et encore…), mais maintenant qu'il s'agit de classes, l'héritage de Python vient à notre secours. Il faut simplement créer une nouvelle classe héritant de `ListView`, qui bénéficie donc, sans duplication de code, de toutes les fonctionnalités de sa classe parente tout en lui ajoutant un filtre particulier basé sur la requête. Bien entendu, cette nouvelle requête ne sera plus écrite dans les URL mais dans les vues de votre application.

Les vues génériques basées sur les classes sont très facilement modifiables, et ceci à plusieurs niveaux. La première méthode, comme nous venons de le voir, consiste à utiliser des paramètres nommés directement à l'appel de la fonction `as_view`, comme vous l'avez fait avec `paginate_by` et `queryset`. La seconde méthode est de créer une classe héritant d'une vue générique. Django rend cela à la fois possible mais surtout très simple. Grâce à quelques méthodes aisément modifiables, vous pouvez presque remplacer intégralement les vues classiques par des vues génériques.

La première méthode que vous pouvez modifier est `get_queryset`. C'est elle qui va calculer la queryset pour le reste de la vue. Comme vous êtes dans une vue générique-classe, vous avez accès à `self` qui contient, entre autre, l'objet `request` et donc l'objet `User` de la requête. Vous pouvez donc écrire votre requête comme suit :

```
class Evenement_Liste(ListView):

    def get_queryset(self):
        events = Evenement.objects.filter(participants = self.request.user,
                                          date__gte = datetime.datetime.now()
                                          )
        return events
```

Dans vos URL, il vous suffit d'invoquer `Event_Listing.as_view()` pour retrouver les événements filtrés par utilisateur.

Vous n'avez rien perdu des capacités de votre vue générique. Pour vous en convaincre, utilisez `paginate_by` et constatez que le comportement de la vue générique reste inchangé.

Complexifiez autant qu'il vous plaira la méthode `get_queryset` ; tant que vous renvoyez une queryset, Django s'en accommodera. Et puisque vous en êtes à installer de nouvelles fonctionnalités, ajoutez encore un champ de recherche pour retrouver rapidement un sous-ensemble d'événements en recherchant dans le titre, la description ou le lieu. L'idéal ici est bien entendu de rester le plus générique possible, donc d'écrire du code qui fonctionne aussi bien pour une recherche dans les dates, dans le titre ou dans la description. Pour commencer, modifiez votre queryset pour chercher sur le titre de vos objets événements. Cette opération se fait en deux temps. Premièrement, créez une nouvelle URL :

```
url(r'^listes/title/(?P<search>[\w-]+)/$',Event_Liste.as_view(paginate_by=10)),
```

Puis, dans la méthode `get_queryset` de la classe `Event_Liste`, accédez au dictionnaire `kwargs` qui contient tous les paramètres fournis à la vue. Filtrez ensuite les résultats si la clé `search` est présente :

```
if "search" in self.kwargs:
    events = events.filter(nom__icontains = self.kwargs['search'])
    return events
```

Vous pouvez bien entendu ajouter ces quelques lignes de code pour chaque champ sur lequel vous souhaitez proposer un filtre, mais il est bien plus intelligent de rendre cette fonctionnalité générique en utilisant le filtre lui-même comme premier argument et le mot à rechercher comme second argument. Pour commencer, changez votre URL :

```
url(r'^liste/(?P<champ>[\w-]+)/(?P<terme>[\w-]+)/
$',Evenement_Liste.as_view(paginate_by=10)),
```

Deux mots-clés sont maintenant capturés par votre URL :

- `champ`, qui correspond au champ sur lequel vous allez filtrer vos résultats (« nom », « description »...) ;
- `terme`, le terme de votre recherche.

Dans votre méthode `get_queryset`, vérifiez maintenant la présence du premier terme et lancez votre recherche comme suit :

```
if 'champ' in self.kwargs:
    events = events.filter((self.kwargs['champ'],self.kwargs['terme']))
```

Comme vous le voyez, en quelques lignes seulement, votre application vient de faire un véritable bond en avant. Vous avez ajouté de nombreuses fonctionnalités avec moins de code, simplement en utilisant la puissance des vues génériques dans Django.

Si vous avez d'autres idées et que vous voulez proposer de nouvelles fonctionnalités, c'est le moment ou jamais ! Django vous offre, avec les nouvelles vues génériques basées sur les classes, un outil simple extrêmement puissant que vous pourrez modeler à l'envi avec très peu de code.

Un peu de nettoyage

Maintenant que votre vue-classe est fonctionnelle, profitez-en pour faire un peu de ménage : supprimez ce qui n'a plus lieu d'être et déplacez ce qui doit l'être.

Tout d'abord, vos URL doivent maintenant ressembler à ceci :

```
from django.conf.urls.defaults import patterns, include, url
from views import create, delete, details, update, delete_participant,
Evenement_Liste
from models import Evenement

urlpatterns = patterns('',
    url(r'^create/$', create),
    url(r'^(\d+)/details/$', details),
    url(r'^liste/$', Evenement_Liste.as_view(paginate_by=10)),
    url(r'^liste/(?P<champ>[\w-]+)/(?P<terme>[\w-]+)/
$',Evenement_Liste.as_view(
            paginate_by=10
            )
        ),
    url(r'^(\d+)/delete/$', delete),
    url(r'^(\d+)/participant/(\d+)/delete/$', delete_participant),
    url(r'^(\d+)/update/$', update),
    )
```

Dans vos vues, supprimez la fonction `liste` qui ne sert plus, puis importez les éléments dont vous avez besoin :

```
from django.views.generic.list import ListView
import datetime
```

Votre classe doit maintenant ressembler à ceci :

```
class Evenement_Liste(ListView):
    def get_queryset(self):
        events = Evenement.objects.filter(
            participants = self.request.user,
            date__gte = datetime.datetime.now()
            )
        if 'champ' in self.kwargs:
            events = events.filter(
                (self.kwargs['champ'],
                 self.kwargs['terme']
                 )
                )
        return events
```

Enfin, supprimez le template `liste.html`, qui ne vous sera plus utile, et déplacez vos templates du dossier `blocks` du dossier `templates` de votre projet à celui de votre application.

Rev 21

Réorganisation des templates.

Afficher les détails d'un événement : DetailView

Nous avons exploré les diverses possibilités des vues génériques listant des collections d'objets. Intéressons-nous maintenant à l'affichage d'un seul objet ; ce n'est pas vraiment différent dans la logique de l'affichage d'une liste d'objets.

La classe qui sert à afficher un objet est `DetailView`. Comme la classe `ListView`, vous pouvez l'utiliser dans vos URL avec la syntaxe suivante :

```
DetailView.as_view(model=<MonModel>)
```

Pour que votre vue générique fonctionne correctement, il faut que vous fournissiez à la méthode `as_view` au minimum un modèle et un identifiant unique, passés dans l'URL, permettant à la classe de retrouver l'objet que vous recherchez. Par défaut, deux identifiants sont recevables.

- `slug_field` : c'est un champ de base de données, de type texte, dont les accents ont été supprimés et les espaces remplacés par des -. C'est un terme très populaire dans la presse. Vous pouvez l'obtenir en définissant dans vos modèles un champ de type `models.Slug_field`.

- `pk` : vous avez déjà rencontré cette écriture, qui signifie *Primary Key* ou clé primaire. C'est bien souvent l'`id` de vos modèles.

Si vous utilisez `pk`, il vous suffit d'écrire une URL du type :

```
url(r'^evenement/(?P<pk>[\d-]+)/$',DetailView.as_view(model = Evenement)),
```

Si vous utilisez `slug_field`, Django cherchera par défaut un champ `slug` dans votre modèle. Si c'est bien ce que vous avez défini dans vos modèles, l'URL suivante suffira :

```
url(r'^evenement/(?P<slug>[\d-]+)/$',DetailView.as_view(model = Evenement)),
```

En revanche, si vous avez nommé votre champ `slug` autrement, vous devez le préciser, par exemple :

```
url(r'^evenement/(?P<titre_slug>[\d-]+)/$',DetailView.as_view(
                                        model = Evenement,
                                        slug_field = titre_slug
                                        )
),
```

Bien entendu, comme pour `ListView`, vous pouvez hériter de la classe `DetailView` pour des besoins plus complexes, qui a le même fonctionnement. Aux méthodes que vous avez découvertes précédemment s'ajoute `get_object` : elle est utilisée pour retrouver l'objet dont vous avez besoin. Par défaut, cette méthode prend en argument `get_queryset`. Ainsi, si vous avez créé une fonction `get_queryset` spécifique, la queryset résultante sera fournie en argument à votre fonction ; dans le cas contraire, c'est `model.objects.all()`, où `model` est le nom du modèle que vous avez défini dans le paramètre `model` de votre fonction `as_view()`.

Conclusion

Cette première approche des vues génériques vous offre un avant-goût des mécanismes avancés de Django qui vous permettront, au fil du temps, de développer de plus en plus vite vos fonctionnalités de base. Vous pourrez ainsi mettre en place rapidement ce que l'on appelle des « produits minimum viables ».

8

Une revue de l'application : factorisation du code

Maintenant que vous avez une compréhension poussée de Django, concernant aussi bien les vues que les templates, il est temps de factoriser le code de votre application pour utiliser pleinement la puissance des vues génériques. Au cours de ce chapitre, vous apprendrez à rendre votre code plus concis et donc moins sujet aux erreurs.

Nous vous conseillons d'effectuer ce type de revue de code aussi souvent que possible. L'effet pervers du développement agile est de développer des fonctionnalités les unes derrière les autres, sans lien entre elles, et de finir par rédiger un code à la qualité douteuse. Ici, vous n'allez pas ajouter de nouvelles fonctionnalités mais sans doute en enlever. Cela va vous aider à définir un flux d'utilisation qui offrira à l'utilisateur une expérience agréable.

Au cours de cette relecture, certaines fonctions vont être réécrites pour les rendre à la fois plus simples, plus concises mais aussi plus performantes. Nous vous proposerons également un squelette CSS qui donnera à votre application une apparence professionnelle pour que vous et vos proches ayez plaisir à l'utiliser. À la fin de ce chapitre, votre application sera définitivement terminée et vous pourrez la mettre en production.

Utiliser les vues génériques dans l'ensemble de votre application

Dans un premier temps, dans cette première revue de code, nous n'utiliserons plus que les vues génériques. Ceci va être relativement simple car, pour le moment, votre interface suit presque littéralement le principe du CRUD.

La vue Evenement_Detail

La vue qui est la plus complexe dans votre application est `Evenement_Detail`. C'est donc par celle-ci que nous allons commencer. Tout d'abord, écrivez une vue très simple :

```
from django.views.generic.detail import DetailView
class Evenement_Detail(DetailView):
    model = Evenement
```

puis l'URL correspondante :

```
from views import Evenement_Detail
    url(r'^(?P<pk>\d+)/detail/$', Evenement_Detail.as_view()),
```

Comme Django vous y invite si vous testez cette vue dans votre navigateur, vous devez écrire le template `personal_calendar/evenement_detail.html`. Pour le moment, contentez-vous de reprendre le template `details.html` que vous avez déjà écrit, en le modifiant pour qu'il corresponde aux besoins de la vue générique :

```
{% extends "base.html" %}

{% block titre %}
    | {{object.nom}}
{% endblock titre %}

{% block content %}
<h1>{{object.nom}} (<a href="/agenda/{{obejct.id}}/update/">Modifier</a>)</h1>
<p>{{object.description}}</p>
<p>le {{object.date}}</p>
<p>à {{object.lieu}}</p>
<div id="participants">
{% for participant in object.evenement_participant_set.all %}
<div class="well">
```

```
{{participant.participant}}: {{participant.get_status_display}}
{% with delete_url=participant.delete_url %}
{% include "personal_calendar/blocks/delete_form.html" %}
{% endwith %}
</div>
{% endfor %}
</div>
{% include "personal_calendar/blocks/participant_form.html" %}
{% with delete_url=object.delete_url %}
{% include "personal_calendar/blocks/delete_form.html" %}
{% endwith %}
{% endblock content %}
```

Testez cette nouvelle vue dans votre navigateur et validez que tout s'affiche bien. Pourtant, plus rien de la logique d'ajout et de suppression d'un participant ne fonctionne. Comment y remédier ?

Tout d'abord, si votre ancienne vue renvoyait un formulaire, ce n'est plus le cas maintenant. En écrivant une méthode `get_context_data`, vous enrichissez le contexte de votre vue et pouvez ainsi à nouveau fournir le formulaire :

```
    def get_context_data(self, **kwargs):
        context = super(Evenement_Detail, self).get_context_data(**kwargs)
        form = Evenement_ParticipantForm(initial = {'evenement':self.object})
        participants = [user.pk for user in self.object.participants.all()]
        form.fields['participant'].queryset=User.objects.exclude(
            pk__in = participants)
        form.fields['evenement'].widget = HiddenInput()
        context['form'] = form
        return context
```

Comme vous le remarquez, l'événement que vous traitez est immédiatement disponible via l'attribut `self.object` de votre classe. Il vous suffit alors d'enrichir le `Context`. Créez une nouvelle méthode qui cherchera à valider le formulaire s'il est soumis en utilisant `POST`. Très logiquement, cette nouvelle méthode se nommera `post()` :

```
    def post(self, request, *args, **kwargs):
        form = Evenement_ParticipantForm(self.request.POST)
        if form.is_valid():
            form.save()
            if self.request.is_ajax():
                delete_form = render_to_string(
                    "personal_calendar/blocks/delete_form.html",
                    {'delete_url': form.instance.delete_url(),
                     },
```

```
                    RequestContext(request)
                    )
                data = {'participant': form.instance.participant.username,
                       'get_status_display': form.instance.get_status_display(),
                        'delete_form': delete_form}
                return HttpResponse(
                    json.dumps(data),
                    mimetype="application/json"
                    )
            else:
                # Pas d'ajax
                return HttpResponseRedirect('/agenda/%s/detail/' % id)
        else:
            # Formulaire invalide
            participants = [user.pk for user in
form.instance.evenement.participants.all()]
            form.fields['participant'].queryset=User.objects.exclude(
                pk__in = participants)
            form.fields['evenement'].widget = HiddenInput()
            if request.is_ajax():
                return render(
                    request, 'personal_calendar/blocks/participant_form.html',
                    {'event':form.instance.evenement,
                     'form': form})
            else:
                return render(request,'personal_calendar/event/details.html',{
                        'event':form.instance.evenement,
                        'form':form})
```

Nous revenons cette fois sur une vue assez classique. Remarquez que l'on doit dupliquer le code d'affichage du formulaire pour chacune des vues. Ce n'est pas optimal…

Factorisation du formulaire

Dans un cas comme celui-ci, il vous faut déporter cette logique dans le formulaire lui-même. Reprenez donc votre fichier `forms.py` et faites en sorte que le champ `evenement` soit caché par défaut :

```
class Evenement_ParticipantForm(ModelForm):
    class Meta:
        model = Evenement_Participant
    def __init__(self,*args, **kwargs):
        super(Evenement_ParticipantForm, self).__init__(*args, **kwargs)
        self.fields['evenement'].widget = HiddenInput()
```

Il faut ensuite que le champ `participant` ne propose que ceux qui ne sont pas encore inscrits à l'événement. Pour cela, vous devez avoir accès à la liste des participants déjà

enregistrés sur ce dernier. C'est plus simple à faire qu'il n'y paraît, car vous utilisez l'instruction suivante pour afficher votre formulaire :

```
form = Evenement_ParticipantForm(initial = {'evenement':self.object})
```

Il vous faut donc récupérer la valeur de `initial['evenement']` pour récupérer l'événement sur lequel vous travaillez et, de fait, les participants qui y sont déjà inscrits. Votre formulaire devient donc le suivant :

```
class Evenement_ParticipantForm(ModelForm):
    class Meta:
        model = Evenement_Participant
    def __init__(self,*args, **kwargs):
        super(Evenement_ParticipantForm, self).__init__(*args, **kwargs)
        self.fields['evenement'].widget = HiddenInput()
        if 'evenement' in self.initial:
            participants = [user.pk for user in
self.initial['evenement'].participants.all()]
            self.fields['participant'].queryset=User.objects.exclude(
                pk__in = participants)
```

Vous utilisez ici un branchement conditionnel avec `if 'evenement' in self.initial:`, car dans le cas de la validation du formulaire, l'attribut `self.initial['evenement']` n'existe pas.

Maintenant que votre formulaire contient toute la logique dont vous avez besoin, allégez votre vue `details` :

```
class Evenement_Detail(DetailView):
    model = Evenement

    def get_context_data(self, **kwargs):
        context = super(Evenement_Detail, self).get_context_data(**kwargs)
        form = Evenement_ParticipantForm(initial = {'evenement':self.object})
        context['form'] = form
        return context

    def post(self, request, *args, **kwargs):
        form = Evenement_ParticipantForm(self.request.POST)
        if form.is_valid():
            form.save()
            if self.request.is_ajax():
                delete_form = render_to_string(
                    "personal_calendar/blocks/delete_form.html",
                    {'delete_url': form.instance.delete_url(),
                     },
```

```
                RequestContext(request)
                )
            data = {'participant': form.instance.participant.username,
                   'get_status_display': form.instance.get_status_display(),
                    'delete_form': delete_form}
            return HttpResponse(
                json.dumps(data),
                mimetype="application/json"
                )
        else:
            # Pas d'ajax
            return HttpResponseRedirect('/agenda/%s/detail/' % id)
    else:
        if request.is_ajax():
            return render(
                request, 'personal_calendar/blocks/participant_form.html',
                {'event':form.instance.evenement,
                 'form': form})
        else:
            return render(request,'personal_calendar/event/details.html',{
                    'event':form.instance.evenement,
                    'form':form})
```

Votre fonction `details` devient alors obsolète et vous pouvez donc la supprimer à la fois des URL et de votre fichier `views.py`.

Des URL DRY

Pourtant, lorsque vous tentez d'accéder à votre vue depuis la page `liste`, vous obtenez une erreur de la part de Django. En effet, ce dernier exécute la méthode `get_absolute_url` de la classe `Evenement`. Or, cette méthode est pour le moment écrite comme suit :

```
def get_absolute_url(self):
    return "/agenda/%s/details" % self.id
```

Dans vos URL, en supprimant la ligne suivante :

```
url(r'^(\d+)/details/$', details),
```

et en la remplaçant par celle-ci :

```
url(r'^(?P<pk>\d+)/detail/$', Evenement_Detail.as_view()),
```

Django ne peut plus trouver d'URL se terminant par `/details/`. La solution qui vient tout de suite à l'esprit consiste à modifier :

```
def get_absolute_url(self):
        return "/agenda/%s/details" % self.id
```

par :

```
    def get_absolute_url(self):
        return "/agenda/%s/detail" % self.id
```

Et, effectivement, vous pouvez tester cette méthode et vous apercevoir que cela fonctionnera tout à fait correctement. Pourtant, ce n'est pas la meilleure façon de procéder. Modifier ses URL et ses modèles pour une seule chose, ce n'est pas DRY, car cela oblige à maintenir les changements à deux endroits différents. Heureusement, il existe une méthode pour ne modifier que les URL et pour que ces changements soient immédiatement pris en compte partout dans l'application.

Tout d'abord, pour que cette méthode fonctionne, nommez vos URL. Ainsi :

```
url(r'^(?P<pk>\d+)/detail/$', Evenement_Detail.as_view()),
```

devient :

```
    url(r'^(?P<pk>\d+)/detail/$', Evenement_Detail.as_view(), name='details'),
```

Dans vos modèles, vous ne renverrez plus une chaîne de caractères représentant l'URL elle-même, mais seulement son nom, avec les paramètres nécessaires à la reconstruction de ladite URL. Cela est possible grâce à `reverse`. Dans votre fichier `models.py`, il vous faut donc l'importer :

```
from django.core.urlresolvers import reverse
```

Ensuite, utilisez-la dans votre méthode `get_absolute_url` :

```
    def get_absolute_url(self):
        return reverse('details', kwargs={'pk':self.pk})
```

Il est bien entendu tout à fait opportun de faire de même pour la méthode `delete_url`. Essayez à titre d'exercice, sachant que si vous éprouvez des difficultés, vous pouvez toujours récupérer l'ensemble du code fonctionnel à la révision 22.

Rev 22

Une vue `details` générique, un formulaire factorisé, des URL DRY.

Créer, modifier, supprimer des événements

La vue `details` était sans aucun doute la plus complexe que vous aviez à écrire. Maintenant, celles qui vous restent sont beaucoup plus simples.

La vue CreateView : création d'un nouvel événement

Pour commencer, il va falloir écrire une vue pour la création d'un nouvel événement. Toujours en utilisant les vues génériques de Django, son écriture est limpide.

Dans vos URL, écrivez :

```
from models import Evenement
from django.views.generic.edit import CreateView
    url(r'^create/$', CreateView.as_view(model=Evenement)),
```

Comme à son habitude, si vous testez cette vue, Django vous avertira qu'il ne peut trouver le template `personal_calendar/evenement_form.html`. Vous avez le choix soit de suivre la convention et de créer le template, soit de spécifier un autre nom de template avec l'argument `template_name`. Pour cet exemple, nous utiliserons la configuration par défaut.

```
{% extends "base.html" %}

{% block titre %}
    | Créer un événement
{% endblock titre %}
{% block content %}
<div class="well">
<h1>{% if form.initial.nom %} {{form.initial.nom}} {%else%}Nouvel événement
{%endif%}</h1>
<form action="" method="POST" class="form-horizontal">
{% csrf_token %}
{{form.as_p}}
<input type="submit" value="{% if form.initial.nom
%}Modifier{%else%}Créer{%endif%}" />
</form>
</div>
{% endblock content %}
```

Comme pour la fonction `create`, les participants sont affichés, mais vous ne pouvez pas valider le formulaire en l'état. Vous aviez réglé ce problème en créant un formulaire spécial `EventForm`, qui n'affichait pas les participants. Heureusement, la vue générique `CreateView` vous permet de préciser un formulaire. Reprenez donc votre fichier `urls.py` et modifiez-le comme suit.

```
from forms import EventForm
    url(r'^create/$', CreateView.as_view(model=Evenement,
form_class=EventForm)),
```

Rien de plus n'est nécessaire pour que votre vue soit maintenant fonctionnelle. Supprimez la fonction `create` et son `import` dans vos URL.

Rev 23

La vue générique `CreateView`.

La vue UpdateView : mettre à jour un événement

La vue `UpdateView` sert à mettre à jour un événement. Elle remplacera la vue-fonction `update` écrite précédemment.

Tout d'abord, importez la vue générique dans vos URL :

```
from django.views.generic.edit import CreateView, UpdateView
```

Puis utilisez cette nouvelle classe dans vos URL :

```
    url(r'^(?P<pk>\d+)/update/$', UpdateView.as_view(model=Evenement,
form_class=EventForm)),
```

Cette fois, il n'est pas nécessaire d'écrire un nouveau template, car Django va réutiliser le template de création écrit précédemment. Il est également inutile de s'occuper de la validation puisque Django s'en charge pour vous. La fonction `update` ne vous est plus d'aucune utilité. Vous pouvez tester la nouvelle vue et vérifier son parfait fonctionnement.

La vue DeleteView : supprimer un événement

Enfin, et cela terminera cette vue d'ensemble des vues génériques, il vous reste à utiliser la vue `DeleteView` qui se chargera de supprimer un événement, en remplaçant la fonction `delete`. Pour que cette vue fonctionne, vous devez lui préciser le modèle de l'objet qu'elle doit supprimer et une URL vers laquelle rediriger l'utilisateur une fois l'objet supprimé.

Comme vous l'avez appris avec `get_absolute_url`, il est possible de nommer les URL pour que Django puisse les retrouver aisément. Comme vous souhaitez rediriger l'utilisateur vers la liste des événements, nommez l'URL de liste comme suit :

```
    url(r'^liste/$', Evenement_Liste.as_view(paginate_by=10), name='liste'),
```

Importez ensuite la classe `DeleteView` :

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
```

Enfin, écrivez votre URL comme ceci :

```
    url(r'^(\d+)/delete/$', DeleteView.as_view(
            model=Evenement,
            success_url=reverse_lazy('liste')
            )
        ),
```

Modifiez également votre template `evenement_detail` pour y inclure un lien vers la suppression de l'objet, en utilisant la syntaxe qui permet à Django de calculer seul l'URL :

```
<a href="{% url delete pk=object.pk %}">Supprimer cet événement</a>
```

Quand vous testez cette URL, Django vous signale qu'il ne trouve pas le template `personal_calendar/evenement_confirm_delete.html`. Comme vous pouvez vous en douter, ce template demande une confirmation à l'utilisateur sur la suppression de l'objet. Vous pouvez donc écrire ce template en utilisant la syntaxe suivante :

```
<p>Êtes-vous certain de vouloir supprimer l'événement {{object.nom}} ? </p>

<form method="POST" action="">
{% csrf_token %}
<input type="submit" value="Oui" /></li>
</form>
<a href="{{ object.get_absolute_url }}">Non</a>
```

Conclusion

Maintenant que vous avez converti votre application à l'utilisation des vues génériques, vous devez avoir un fichier `urls.py` comme ceci :

```
from django.conf.urls.defaults import patterns, include, url
from models import Evenement
from forms import EventForm
from views import delete_participant, Evenement_Liste, Evenement_Detail
from django.core.urlresolvers import reverse_lazy
from django.views.generic.edit import CreateView, UpdateView, DeleteView
```

```
urlpatterns = patterns('',
    url(r'^create/$', CreateView.as_view(
            model=Evenement,
            form_class=EventForm)
        name='create'),
    url(r'^(?P<pk>\d+)/detail/$', Evenement_Detail.as_view(), name='details'),
    url(r'^liste/$', Evenement_Liste.as_view(paginate_by=10), name='liste'),
    url(r'^liste/(?P<champ>[\w-]+)/(?P<terme>[\w-]+)/
$',Evenement_Liste.as_view(
            paginate_by=10, name='liste_filtre'
            )
        ),
    url(r'^(?P<pk>\d+)/delete/$', DeleteView.as_view(
            model=Evenement,
            success_url=reverse_lazy('liste')
            ),
        name='delete'
        ),
    url(r'^(\d+)/participant/(\d+)/delete/$', delete_participant),
    url(r'^(?P<pk>\d+)/update/$', UpdateView.as_view(
            model=Evenement,
            form_class=EventForm),
        name='modifier'),
    )
```

Votre fichier `views.py`, lui, a connu un régime amaigrissant particulièrement efficace. En effet, il ne doit plus contenir que deux classes et une vue-fonction :

```
from forms import EventForm, Evenement_ParticipantForm
from models import Evenement, Evenement_Participant
from django.shortcuts import render, HttpResponseRedirect, render_to_response
from django.http import HttpResponse
from django.contrib.auth.models import User
import json
from django.template.loader import render_to_string
from django.template import RequestContext
from django.views.generic.list import ListView
from django.views.generic.detail import DetailView
import datetime


class Evenement_Detail(DetailView):
    model = Evenement

    def get_context_data(self, **kwargs):
        context = super(Evenement_Detail, self).get_context_data(**kwargs)
```

```
        form = Evenement_ParticipantForm(initial = {'evenement':self.object})
        context['form'] = form
        return context

    def post(self, request, *args, **kwargs):
        form = Evenement_ParticipantForm(self.request.POST)
        if form.is_valid():
            form.save()
            if self.request.is_ajax():
                delete_form = render_to_string(
                    "personal_calendar/blocks/delete_form.html",
                    {'delete_url': form.instance.delete_url(),
                     },
                    RequestContext(request)
                    )
                data = {'participant': form.instance.participant.username,
                       'get_status_display': form.instance.get_status_display(),
                        'delete_form': delete_form}
                return HttpResponse(
                    json.dumps(data),
                    mimetype="application/json"
                    )
            else:
                # Pas d'ajax
                return HttpResponseRedirect('/agenda/%s/detail/' % id)
        else:
            if request.is_ajax():
                return render(
                    request, 'personal_calendar/blocks/participant_form.html',
                    {'event':form.instance.evenement,
                     'form': form})
            else:
                return render(request,'personal_calendar/event/details.html',{
                        'event':form.instance.evenement,
                        'form':form})

class Evenement_Liste(ListView):
    def get_queryset(self):
        events = Evenement.objects.filter(
            participants = self.request.user,
            date__gte = datetime.datetime.now()
            )
        if 'champ' in self.kwargs:
            events = events.filter(
                (self.kwargs['champ'],
                 self.kwargs['terme']
                 )
                )
        return events
```

```
def delete_participant(request, id, participant):
    if request.method == "POST":
        evenement = Evenement.objects.get(pk = id)
        participant = User.objects.get(pk = participant)
        a_supprimer = Evenement_Participant.objects.get(
                      evenement = evenement,
                      participant = participant
                      )
        a_supprimer.delete()
        if request.is_ajax():
            return HttpResponse("OK")
    return HttpResponseRedirect('/agenda/%s/details/' % id)
```

Vous retrouverez l'ensemble du code à la révision 24.

Rev 24

Les vues génériques intégrées.

9

Les notions de partage

Actuellement, Django vous propose la liste complète des utilisateurs lorsque vous voulez procéder à un enregistrement sur un événement. Si l'application est utilisée dans un cadre amical ou familial restreint, cela ne pose pas de problème. En revanche, dès qu'elle compte plus d'une trentaine d'utilisateurs, cela commence à devenir problématique. Nous allons donc créer un carnet d'adresses et ajouter des notions de partage à notre application.

Pour faciliter l'enregistrement des utilisateurs, vous allez adjoindre un carnet d'adresses à votre projet. Pour cela, ajoutez l'application `addressbook` :

```
python manage.py startapp addressbook
```

Le besoin : gérer sa propre liste d'utilisateurs connectés

Il s'agit de donner le moyen à chaque utilisateur de gérer sa propre liste d'utilisateurs connectés. Il pourra ainsi envoyer des invitations à des contacts qui ne sont pas encore utilisateurs de son projet Django et les manipuler comme il le souhaite : ajouter ou soustraire des informations complémentaires, par exemple, mettre à jour les données de son carnet d'adresses en fonction des renseignements que l'utilisateur connecté aura ou non mis à jour, etc.

Spécification des modèles

Cette nouvelle application, si elle se base sur le modèle `User` de l'application `django.contrib.auth`, aura tout de même besoin de modèles supplémentaires. Tout d'abord, il en faut un qui stocke les utilisateurs présents dans le carnet d'adresses ; nous l'appellerons `Contact`. Ensuite, il en faut un autre qui stocke les informations complémentaires qu'un utilisateur aura renseignées à propos d'un autre utilisateur. Les données du contact dans le carnet d'adresses ne seront pas partagées avec l'utilisateur en question, mais elles resteront privées.

Définition des modèles

Les modèles seront donc définis comme suit :

```
class Contact(models.Model) :
"""
Un contact est une entrée du carnet d'adresses.
"""
# L'utilisateur à qui appartient le contact :
owner = models.ForeignKey(User)
# L'utilisateur Django auquel ce contact fait référence :
user = models.ForeignKey(User, related_name='friend')
# Enregistrons également les données relatives aux invitations.
# Une invitation a-t-elle été émise ?
invitation_send = models.BooleanField()
# L'invitation a-t-elle été acceptée ?
invitation_accepted = models.BooleanField()
optional_informations = models.OneToOne(UserInfo,blank=True,null=True)
```

Vous constatez que nous faisons appel à un modèle `UserInfo` qui n'existe pas encore. Dans celui-ci, il est possible de stocker toutes les informations complémentaires que vous souhaitez rendre disponibles à l'édition dans le carnet d'adresses. Pour faire simple, nous allons simplement définir des cercles, dans le goût de Google+, mais rien ne vous empêche de nourrir le modèle de vos propres idées.

Chaque contact appartiendra donc à un cercle, c'est-à-dire une catégorie, qui permettra à l'utilisateur de « ranger » ses contacts. Vous allez également réserver un champ pour écrire quelques notes sur ce contact. Le modèle `UserInfo` sera donc très simple :

```
class UserInfo(models.Model):
    circle = models.ManyToManyField(Circle)
    notes = models.TextField()
```

Nous faisons appel à un nouveau modèle, `Circle`, et nous définissons un many-to-many car il doit être possible de placer un contact dans plusieurs cercles à la fois. Comme un utilisateur peut être ajouté à de multiples cercles par de multiples utilisateurs, nous allons dans le modèle `Circle` définir la personne qui a créé le cercle. Cela vous simplifiera la tâche quand il s'agira d'afficher les carnets d'adresses, en vous évitant des requêtes coûteuses dans la base de données.

```
    class Circle(models.Model):
        name = models.CharField(max_length=250)
        owner = models.ForeignKey(User)
```

Les requêtes courantes

Une bonne habitude consiste à créer dans vos modèles des méthodes qui correspondent aux requêtes les plus courantes.

Dans votre modèle Contact

```
    def all_contacts(self,user):
        """
        retrouve tous les contacts d'un utilisateur
        """
        return Contact.objects.filter(owner=user)
    def all_circles(self,user):
        """
        retrouve tous les cercles qu'un utilisateur a défini
        """
        return Circle.objects.filter(owner=user)
```

Dans votre modèle Circle

```
    def contacts(self):
        """
        renvoie l'ensemble des utilisateurs appartenant au cercle
        """
        return self.user_info.contact_set.all()

    def is_in_circle(self,user):
        """
        renvoie True si l'utilisateur est dans le cercle, False dans le cas
contraire
        """
        if user in self.user_info.contact_set.all():
            return True
        return False
```

Testez ces quelques lignes en utilisant le shell Python. Par exemple, pour la méthode `all_contacts` de la classe `Contact`, créez un objet `contact` et saisissez les lignes suivantes :

```
>>> from addressbook.models import Contact
>>> me = Contact.objects.all()[0]
>>> from django.contrib.auth.models import User
>>> a = User.objects.get(pk=1)
>>> Contact().all_contacts(a)
[<Contact: Contact object>]
```

Gestion des invitations

Afin de rendre votre application fonctionnelle, la première chose que vous allez devoir faire, c'est de gérer les invitations. Il vous faut permettre à un utilisateur d'ajouter de nouveaux contacts à son carnet d'adresses. Pour cela, ils doivent être présents dans la base de données des utilisateurs. Deux cas de figure sont donc envisageables.

- Dans le premier cas, le plus simple, l'utilisateur que vous tentez d'ajouter à votre carnet d'adresses est déjà présent dans la base de données ; vous n'avez donc plus qu'à ajouter un nouvel objet `contact` et à envoyer un e-mail à l'utilisateur en question pour lui signifier qu'il a été ajouté à un carnet d'adresses.
- Le second cas concerne un utilisateur qui n'est pas présent dans la base de données des utilisateurs. Il faut alors envoyer un e-mail à cette personne en lui proposant de se créer un compte sur l'application. Lorsque c'est fait, l'objet `contact` peut être créé.

La logique de ce cheminement utilise le champ `email` d'un utilisateur Django. Or, votre application `usermanagement` ne vous permet pas aujourd'hui de définir un e-mail lors de la création ; il vous faut donc corriger cela.

Ajouter le champ email à la création d'un nouvel utilisateur

Dans votre application `usermanagement`, créez un nouveau fichier `forms`. Il définira une nouvelle classe de formulaire, `UserCreateForm`, qui va hériter du formulaire de base de création d'utilisateur `UserCreationForm` et lui ajouter un champ `email` (au niveau de la base de données, vous n'avez pas de modification à effectuer car le champ `email` est déjà présent sur l'objet `User`). Votre nouveau formulaire va donc ajouter le champ `email` au formulaire et définir une méthode `save` qui le sauvegardera dans la base de données :

```
from django.contrib.auth.forms import UserCreationForm
from django import forms

class UserCreateForm(UserCreationForm):
    email = forms.EmailField(required=True)
```

```
    def save(self, commit=True):
        user = super(UserCreateForm, self).save(commit=False)
        user.email = self.cleaned_data["email"]
        if commit:
            user.save()
        return user
```

Il faut ensuite, dans les vues de l'application `usermanagement`, que vous utilisiez ce nouveau formulaire en lieu et place de l'ancien :

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from forms import UserCreateForm

def create_account(request):
    if request.method == "POST":
        form = UserCreateForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect('/user/succes/')
    else:
        form = UserCreateForm()
    return render(request,'user/create.html',{'form':form})
```

Le modèle Invitation

Maintenant que vos utilisateurs Django possèdent un champ `email`, revenez sur votre application `addressbook` et concentrez-vous sur le modèle `Invitation`. Ce dernier doit contenir l'adresse électronique de la personne invitée et un lien vers l'utilisateur qui effectue l'invitation :

```
class Invitation(models.Model):
    email = models.EmailField()
    sender = models.ForeignKey(User)

    def __unicode__(self):
        return self.email

    def get_absolute_url(self):
        return reverse('invitation_list')

    class Meta:
        unique_together = ('email', 'sender')
```

Cette classe définit une méthode __unicode__ qui identifie une invitation par email, ainsi qu'un get_absolute_url en utilisant reverse vers l'URL qui liste toutes les invitations, invitation_list, et s'assure qu'une seule invitation existe avec la même adresse électronique et le même expéditeur. Comme l'URL invitation_list n'existe pas encore, il vous faut l'écrire.

1 Dans vos vues, définissez une vue-classe qui hérite de ListView, qui renvoie l'ensemble des invitations dont l'expéditeur est l'utilisateur actuellement connecté. Comme vous avez évidemment besoin du modèle Invitation, importez-le en début de fichier :

```
from models import Invitation
```

2 Définissez votre nouvelle classe :

```
class InvitationListView(ListView):
    def get_queryset(self):
        return Invitation.objects.filter(sender=self.request.user)
```

3 Vous pouvez maintenant l'utiliser dans vos URL :

```
from views import InvitationListView
    url(r'^invitations/$', InvitationListView.as_view(),
name="invitation_list")
```

Le formulaire d'invitation

Le formulaire d'invitation, tel qu'il est créé par une CreateView par exemple, présente à l'utilisateur un champ email et un champ sender qui est une liste de tous les utilisateurs de la base de données. Seulement, pour que le système d'invitation fonctionne, vous ne devez pas lui permettre de choisir qui envoie l'invitation ; ce doit toujours être l'utilisateur actuellement connecté.

Créez un fichier forms dans votre application addressbook et ajoutez le formulaire :

```
from django.forms import ModelForm
from models import Invitation

class InvitationForm(ModelForm):

    class Meta:
        model = Invitation
        exclude = ('sender',)
```

Maintenant que le champ sender n'est plus affiché, vous devez vous assurer qu'il est bien enregistré. Par ailleurs, il faut que vous validiez vous-même l'unicité définie

dans les modèles, car Django n'a plus accès au champ `sender`. Dans vos vues, définissez la classe `InvitationView` comme suit :

```
class InvitationView(CreateView):
    form_class = InvitationForm
    model = Invitation

    def form_valid(self, form):
        obj = form.save(commit=False)
        obj.sender = self.request.user
        try:
            Invitation.objects.get(email=obj.email, sender=obj.sender)
            form._errors["email"] = ErrorList(
                [u"Une invitation a déjà été envoyée à cette adresse."])
            return super(InvitationView, self).form_invalid(form)
        except Invitation.DoesNotExist:
            pass
        obj.save()
        return HttpResponseRedirect(obj.get_absolute_url())
```

Pour tester ce code, définissez une URL qui permette d'y accéder :

```
    url(r'^invitation/$', InvitationView.as_view()),
```

Envoi des e-mails avec Django

Il faut maintenant savoir si l'utilisateur existe dans la base de données de Django. Si tel est le cas, il faut créer immédiatement le contact et envoyer un e-mail ; sinon, il faut envoyer un e-mail proposant à ce nouvel utilisateur de se créer un compte. Dans le framework Django, ce type d'envoi est particulièrement simplifié.

Dans un environnement de production, vous souhaiterez sans doute utiliser un serveur SMTP. Dans votre fichier `prod_settings.py`, indiquez le *back-end* `SMTP` pour l'envoi d'e-mails :

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

Puis précisez les données du serveur. Par exemple, si vous utilisez le SMTP de Gmail, vos données de compte seront les suivantes :

```
EMAIL_USE_TLS = True
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_HOST_USER = 'votreemail@gmail.com'
EMAIL_HOST_PASSWORD = 'votremotdepasse'
EMAIL_PORT = 587
```

Bien entendu, pendant le développement, vous n'avez certainement pas envie d'envoyer des e-mails par dizaines. Dans ce cas, Django vous offre la possibilité de les écrire dans votre terminal en utilisant le back-end `Console`. Dans votre fichier `local_settings.py`, écrivez :

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Utilisez ensuite l'une des méthodes d'envoi proposées par Django. Dans l'exemple, nous utilisons `send_mail` qui vous est fourni par `django.core.mail import send_mail`. Sa syntaxe est la suivante :

```
send_mail(titre,expéditeur,[liste de destinataires], fail_silently=True ou
False)
```

Maintenant, il vous faut gérer deux cas : soit l'e-mail de l'invitation correspond à un utilisateur présent dans la base de données, soit il n'existe pas encore. Vérifiez que l'utilisateur est inscrit dans la base à l'aide de la requête suivante :

```
User.objects.get(email=invitation.email)
```

Dans le cas où l'utilisateur n'est pas trouvé, proposez dans votre message un lien vers le formulaire de création de compte que vous avez écrit dans l'application `usermanagement`. Pour cela, vous devez connaître l'URL de cette vue, qui sera forcément différente selon que vous soyez en production ou en développement. Pour régler intelligemment ce problème, utilisez une application des contribs de Django :

```
from django.contrib.sites.models import Site
```

Elle va fournir le domaine de votre site grâce à la méthode :

```
Site.objects.get_current().domain
```

Pour que cette méthode fonctionne, rendez-vous sur `http://localhost:8000/admin/sites/site/1/` et remplissez le champ `nom de domaine`. En développement, il aura la valeur `http://localhost:8000`. En production, cela dépendra évidemment du nom de domaine que vous avez choisi.

Maintenant que toutes les pièces du puzzle sont en place, écrivez dans vos vues la fonction `send_invitation` :

```
def send_invitation(invitation):
    try:
        User.objects.get(email=invitation.email)
```

```
        message = "{0} vous a ajouté à ses contacts".format(
            invitation.sender.username)
    except User.DoesNotExist:
        message = """
{0} vous invite à rejoindre ses contacts.
inscrivez-vous sur http://{1}/user/create_account/ pour accepter son
invitation.
""".format(invitation.sender.username, Site.objects.get_current().domain)

    send_mail('Une invitation vous a été envoyée',
              message,
              invitation.sender,
              [invitation.email],
              fail_silently=False
              )
```

Puis ajoutez un appel à cette fonction dans la méthode `form_valid` de votre classe `InvitationView` :

```
class InvitationView(CreateView):
    form_class = InvitationForm
    model = Invitation

    def form_valid(self, form):
        obj = form.save(commit=False)
        obj.sender = self.request.user
        try:
            Invitation.objects.get(email=obj.email, sender=obj.sender)
            form._errors["email"] = ErrorList(
                [u"Une invitation a déjà été envoyée à cette adresse"])
            return super(InvitationView, self).form_invalid(form)
        except Invitation.DoesNotExist:
            pass
        obj.save()
        send_invitation(obj)
        return HttpResponseRedirect(obj.get_absolute_url())
```

Créer le contact dans le carnet d'adresses de l'utilisateur

Il reste maintenant à créer effectivement le contact dans le carnet d'adresses de l'utilisateur. Si ce dernier est déjà présent dans la base de données, vous pouvez immédiatement le créer :

```
contact = Contact(owner=invitation.sender, user = user)
contact.save()
invitation.delete()
```

En revanche, si l'utilisateur n'existe pas, l'invitation reste en l'état. C'est seulement au moment où l'utilisateur invité ouvrira son compte que vous pourrez créer le contact. Pour cela, vous devez employer les signaux.

Les signaux

Dans le cas présent, nous souhaitons exécuter des instructions lorsqu'un nouvel utilisateur est créé. Seulement, vous n'avez pas accès à ce modèle et vous ne pouvez donc pas créer une méthode `save` sur le modèle `User` de Django. Heureusement, ce framework vous informe néanmoins de la création d'un nouvel utilisateur grâce aux signaux. Ces derniers servent à détecter la création d'un nouvel objet et à réaliser une action de votre choix.

Ici, comme nous souhaitons effectuer une action après que l'objet `User` a été créé, utilisez le signal `post_save`. Importez cette fonction dans vos modèles :

```
from django.db.models.signals import post_save
```

Puis, définissez une fonction qui se chargera de créer les contacts nécessaires :

```
def create_contact_on_user_create(sender, instance, created, **kwargs):
    if created == True:
        try:
            invitations = Invitation.objects.filter(email=instance.email)
            for invitation in invitations:
                contact = Contact(owner=invitation.sender, user=instance)
                contact.save()
                invitation.delete()
        except Invitation.DoesNotExist:
            pass
```

Enfin, connectez cette fonction au modèle `User` :

```
post_save.connect(create_contact_on_user_create, sender=User)
```

Rediriger vers la fiche contact et non l'invitation

Votre gestion des invitations est maintenant pratiquement fonctionnelle. Il reste cependant un point qu'il vous faut traiter. En effet, en testant l'application, vous avez peut-être constaté que lorsque l'utilisateur existe déjà, il est redirigé vers une liste d'invitations qui est vide ou, tout du moins, qui ne contient pas l'invitation qu'il vient tout juste d'envoyer. C'est assez logique. En effet, comme le contact est créé, l'invitation n'a plus lieu d'être et a donc été supprimée. Cependant, pour l'utilisateur, c'est assez perturbant. Afin de corriger cela, il suffit de rediriger l'utilisateur sur le contact et non sur l'invitation.

Un contact doit donc avoir sa propre URL :

```
url(r'^contact/(?P<pk>\d+)/$', DetailView.as_view(model=Contact),
name="contact_detail"),
```

Bien entendu, écrivez la fiche contact. Pour le moment, comme nous nous occupons seulement de la logique de l'application, le template va être particulièrement succinct :

contact_detail.html

```
{{object}}
```

Maintenant, écrivez votre méthode `get_absolute_url` sur votre objet `Contact` :

```
    def get_absolute_url(self):
        return reverse('contact_detail', kwargs={'pk':self.pk})
```

Modifiez vos vues pour l'utiliser :

```
class InvitationView(CreateView):
    form_class = InvitationForm
    model = Invitation

    def form_valid(self, form):
        obj = form.save(commit=False)
        obj.sender = self.request.user
        try:
            Invitation.objects.get(email=obj.email, sender=obj.sender)
            form._errors["email"] = ErrorList(
                [u"Une invitation a déjà été envoyée à cette adresse"])
            return super(InvitationView, self).form_invalid(form)
        except Invitation.DoesNotExist:
            pass
        obj.save()
        return send_invitation(obj)

def send_invitation(invitation):
    try:
        user = User.objects.get(email=invitation.email)
        message = "{0} vous a ajouté à ses contacts".format(
            invitation.sender.username)
        contact = Contact(owner=invitation.sender, user = user)
        contact.save()
        invitation.delete()
        return HttpResponseRedirect(contact.get_absolute_url())
    except User.DoesNotExist:
        message = """
```

```
{0} vous invite à rejoindre ses contacts.
inscrivez vous sur http://{1}/user/create_account/ pour accepter son invitation.
""".format(invitation.sender.username, Site.objects.get_current().domain)

    send_mail('Une invitation vous a été envoyée',
              message,
              invitation.sender,
              [invitation.email],
              fail_silently=False
              )
    return HttpResponseRedirect(invitation.get_absolute_url())
```

Revue de code

Votre gestion des invitations est maintenant terminée et vos utilisateurs vont pouvoir remplir aisément leur carnet d'adresses. Bien entendu, il reste quelques vues à écrire, à gérer les groupes, à permettre à l'utilisateur d'ajouter des notes sur un contact… Néanmoins, ces vues ne recèlent aucune difficulté et ce ne sera pour vous que l'occasion d'une révision. Pour cela, nous vous conseillons de récupérer le code de l'application à la révision 25, puis de tenter par vous-même d'écrire les vues manquantes.

Rev 25

Carnet d'adresses : les invitations.

Les vues

Maintenant, la bonne question à se poser concerne les vues dont vous allez avoir besoin. En premier lieu, bien entendu, il s'agit d'afficher un carnet d'adresses, c'est-à-dire une liste de contacts, puis la liste des cercles que votre utilisateur a définie. Il vous faut également la liste des utilisateurs appartenant à un cercle et, enfin, la fiche d'un utilisateur. À ces vues « génériques » d'affichage de données, vous devez également ajouter les vues de création, de modification et de suppression pour les modèles `Circle` et `Contact`. Nous vous laissons le soin d'écrire vos propres vues, URL et templates.

Si vous êtes perdu ou si vous ne savez pas comment réaliser telle ou telle tâche, téléchargez les exemples du code à la révision 26.

Rev 26

Le carnet d'adresses.

Voici les quelques pièges que vous allez rencontrer et comment les contourner.

Le formulaire d'ajout d'un cercle

Lorsque vous voudrez offrir à un utilisateur la possibilité de créer un cercle, vous ne devez pas afficher le champ `owner` puisque celui-ci doit correspondre à l'utilisateur actuellement connecté :

```
class CircleForm(ModelForm):

    class Meta:
        model = Circle
        exclude = ('owner')
```

Lors de la validation, vous récupérerez l'utilisateur actuellement connecté et sauvegarderez votre nouveau cercle :

```
class CircleCreateView(CreateView):
    form_class = CircleForm
    model = Circle

    def form_valid(self, form):
        obj = form.save(commit=False)
        obj.owner = self.request.user
        obj.save()
        return HttpResponseRedirect(obj.get_absolute_url())
```

Le formulaire de modification d'un contact

Vous ne devez pas laisser un utilisateur modifier un contact, du moins pas directement, car les champs `email` ou `nom` proviennent directement du modèle `User`. En revanche, l'utilisateur peut modifier les informations optionnelles d'un contact, à savoir la note qui lui est attachée et son cercle.

Dans ce formulaire, vous devez laisser une liste de cercles que l'utilisateur peut choisir mais elle ne doit contenir que les cercles que la personne connectée a lui-même créés :

```
class UserInfoUpdateView(UpdateView):

    def get_form(self, form_class):
        form = super(UserInfoUpdateView,self).get_form(form_class)
        form.fields['circle'].queryset =
Circle.objects.filter(owner=self.request.user)
        return form
```

Assurez-vous que, dans vos modèles, un contact est toujours lié à un objet `UserInfo`, et inversement. Pour cela, surclassez `save` et `delete`.

```
    def save(self, *args, **kwargs):
       super(Contact, self).save(*args, **kwargs)
       if self.optional_informations == None:
           infos = UserInfo.objects.create()
           self.optional_informations = infos
           self.save()
           infos.save()

    def delete(self, *args, **kwargs):
        optional_informations = self.optional_informations
        super(Contact, self).delete(*args, **kwargs)
        optional_informations.delete()
```

Relier les applications Calendrier et Agenda

Maintenant que vos deux applications sont écrites, vous allez les faire fonctionner ensemble. Ce sera la dernière itération de votre projet (du moins dans ce livre...). Votre travail va consister en deux choses.

- Tout d'abord, à la création d'un nouvel événement, le formulaire ne devra proposer que les contacts présents dans le carnet d'adresses de l'utilisateur. En tant qu'instigateur de l'événement, il sera immédiatement enregistré comme étant l'hôte de celui-ci. De plus, il ne pourra plus enregistrer des participants qu'avec le statut `invité`.
- Ensuite, les utilisateurs ayant été invités à un événement devront recevoir un message le leur signalant. Ils pourront alors suivre le lien qu'ils auront reçu avec le message et changer leur statut d'`invité` à `confirmé` ou `désisté`. Cette manipulation enverra un e-mail de réponse à l'hôte de l'événement.

Présenter les utilisateurs du carnet d'adresses

Après la création d'un événement, la méthode `get_context_data` de la classe `Evenement_Detail` est appelée à l'ajout d'un nouveau participant. C'est donc à cet endroit que vous devez filtrer les participants pour ne présenter que les membres du carnet d'adresses de l'utilisateur. Cependant, cette liste est déjà filtrée au niveau du formulaire lui-même avec :

```
        if 'evenement' in self.initial:
            participants = [user.pk for user in
self.initial['evenement'].participants.all()]
            self.fields['participant'].queryset=User.objects.exclude(
                pk__in = participants)
```

Dans votre code, récupérez la queryset de `self.fields['participant']` et filtrez-la de nouveau :

```
queryset = form.fields['participant'].queryset
        form.fields['participant'].queryset = queryset.filter(
            contact__owner=self.request.user)
```

Ajouter l'utilisateur créateur de l'événement à la liste des participants

Lorsqu'un utilisateur crée un nouveau participant, vous allez vérifier qu'il est bien dans la liste des participants. Dans la méthode `post` de la classe `Evenement_Detail`, vous pouvez le vérifier avec :

```
if not self.request.user in cleaned_data['evenement'].participants.all():
```

Pour ajouter l'utilisateur, il suffit alors de créer un nouvel objet `Evenement_Participant`. Votre méthode `post` est maintenant comme suit :

```
    def post(self, request, *args, **kwargs):
        form = Evenement_ParticipantForm(self.request.POST)
        if form.is_valid():
            cleaned_data = form.cleaned_data
            if not self.request.user in
cleaned_data['evenement'].participants.all():
                Evenement_Participant.objects.create(
                    evenement=cleaned_data['evenement'],
                    status=0,
                    participant=self.request.user)
            form.save()
...
```

La gestion des statuts

Il vous reste maintenant à modifier les statuts. En effet, un utilisateur invité sera enregistré sous le statut `invité` et le créateur de l'événement ne pourra donc plus choisir lui-même le statut.

Supprimez le statut du formulaire :

```
class Evenement_ParticipantForm(ModelForm):
    class Meta:
        model = Evenement_Participant
        exclude = ('status',)
```

Puis ajoutez-le dans votre méthode `post` :

```
        if form.is_valid():
            obj = form.save(commit=False)
            obj.status = 1
            cleaned_data = form.cleaned_data
            if not self.request.user in
cleaned_data['evenement'].participants.all():
                Evenement_Participant.objects.create(
                    evenement=cleaned_data['evenement'],
                    status=0,
                    participant=self.request.user)
            obj.save()
```

Empêcher la suppression de l'hôte

Maintenant, il vous faut encore effacer le lien `supprimer` pour l'utilisateur créateur de l'événement sur la page `details`. Comme seul l'utilisateur créateur de l'événement possède le statut `hôte`, c'est assez simple à faire, directement sur le template :

```
{% if participant.status != 0 %}
{% with delete_url=participant.delete_url %}
{% include "personal_calendar/blocks/delete_form.html" %}
{% endwith %}
{% endif %}
```

L'envoi d'un e-mail à l'ajout d'un participant

La logique d'ajout d'un participant prend maintenant en compte votre application de carnet d'adresses. Il faut encore envoyer un e-mail à l'utilisateur qui vient d'être invité à un événement. Le contenu du message devra être réécrit dès que vous aurez écrit la fonction qui permettra à un utilisateur de choisir son statut : `confirmé` ou `désisté`.

Pour le moment, il s'agit simplement de savoir où vous allez placer l'envoi de l'e-mail. En toute logique, dès que vous savez que le formulaire d'ajout de participant est valide, vous pouvez envoyer le message. La fonction qui se chargera de cet envoi peut ressembler à ce qui suit :

```
def send_invitation(evenement_participant):
    sender = evenement_participant.evenement.evenement_participant_set.get(
        status=0)

    message = """{0} vous invite à l'événement {1} qui se déroulera le {2}
              à {3}:
              {4}
```

```
""".format(sender.participant,
           evenement_participant.evenement.nom,
           evenement_participant.evenement.date,
           evenement_participant.evenement.lieu,
           evenement_participant.evenement.description)
    sender = sender.participant.email
    subject = "Vous êtes invité par {0}".format(sender)
    recipient_list = [evenement_participant.participant.email]
    send_mail(subject, message,sender,
    recipient_list, fail_silently=False)
```

Vous pouvez l'utiliser dès la validation du formulaire, c'est-à-dire juste après la sauvegarde de votre objet :

```
if form.is_valid():
            obj = form.save(commit=False)
            obj.status = 1
            cleaned_data = form.cleaned_data
            if not self.request.user in
cleaned_data['evenement'].participants.all():
                Evenement_Participant.objects.create(
                    evenement=cleaned_data['evenement'],
                    status=0,
                    participant=self.request.user)
            obj.save()
            send_invitation(obj)
```

Laisser l'utilisateur indiquer son statut

Une fois que l'utilisateur est informé d'une invitation, il doit pourvoir confirmer ou non sa venue. Il faut donc qu'il puisse modifier l'objet `Evenement_Participant` qui le concerne. Pour cela, employez une vue générique `UpdateView` en la modifiant pour générer une URL de redirection. Le code de cette vue pourra être le suivant :

```
class UpdateParticipantView(UpdateView):
    model = Evenement_Participant
    form_class = UpdateParticipantForm

    def form_valid(self, form):
        form.save()
        return HttpResponseRedirect("/agenda/liste/")
```

Il vous faut bien entendu écrire le formulaire `UpdateParticipantForm` qui se chargera de cacher l'événement et l'utilisateur :

```
class UpdateParticipantForm(ModelForm):
    class Meta:
        model = Evenement_Participant
        exclude = ('evenement', 'participant')

    status = ChoiceField(choices=(
                        (2, "désisté"),
                        (3, "confirmé")
                        ))
```

En redéfinissant `status`, vous n'affichez que deux choix possibles : `désisté` et `confirmé`. Vous devez encore mettre à jour le modèle `Evenement_Participant` en y ajoutant le statut `confirmé` qui n'existe pas encore :

```
    status_choices = (
                        (0, "hôte"),
                        (1, "invité"),
                        (2, "désisté"),
                        (3, "confirmé"),
                        )
```

Puis écrivez votre template :

evenement_participant_form.html

```
<form action="" method="POST">
{% csrf_token %}
{{form}}
<input type="submit" value="valider" />
</form>
```

Enfin, écrivez l'URL qui mènera à ce formulaire :

```
url(r'^participation/(?P<pk>\d+)/$', UpdateParticipantView.as_view()),
```

Maintenant que tout est prêt pour gérer la participation d'un utilisateur, reprenez le code de la fonction qui envoie un e-mail aux participants et passez-lui en paramètre l'URL de votre formulaire, en utilisant l'application contribs `site`, comme vous l'avez fait lors de l'envoi d'invitations pour votre carnet d'adresses :

```
def send_invitation(evenement_participant):

    sender = evenement_participant.evenement.evenement_participant_set.get(
```

```
        status=0)

    message = """{0} vous invite à l'événement {1} qui se déroulera le {2}
              à {3}:
              {4}
Indiquez votre participation sur http://{5}/agenda/participation/{6}/
""".format(sender.participant,
           evenement_participant.evenement.nom,
           evenement_participant.evenement.date,
           evenement_participant.evenement.lieu,
           evenement_participant.evenement.description,
           Site.objects.get_current().domain,
           evenement_participant.pk)
    sender = sender.participant.email
    subject = "Vous êtes invité par {0}".format(sender)
    recipient_list = [evenement_participant.participant.email]
    send_mail(subject, message,sender,
    recipient_list, fail_silently=False)
```

Rev 27

Gestion des participants.

Maintenant que votre application est fonctionnelle, elle peut être mise en production sur une véritable machine de production. Pour cela, vous avez besoin de deux choses : un serveur web (comme Apache ou Nginx) et un connecteur Python.

Pour notre exemple, voici les choix techniques que nous avons faits : Gunicorn sera le lien entre le serveur web et le projet Django, et Nginx sera le serveur web. Le projet sera servi au sein d'un environnement virtuel, qu'il faut créer sur le serveur de production :

```
$ virtualenv environnement_de_production
$ source environnement_de_production/bin/activate
```

Téléchargez votre projet sur votre machine de production : via ftp, `ssh` ou, mieux, en clonant votre projet si celui-ci est géré par un gestionnaire de versions. Installez également votre serveur web : `apt-get install nginx` sous Debian, par exemple.

Maintenant que votre projet est sur votre serveur de production, installez toutes les dépendances :

```
$ pip install -r requirement.txt
```

Installez également Gunicorn qui va permettre de faire le lien entre votre projet et votre serveur :

```
$ pip install gunicorn
```

Testez enfin que votre application fonctionne correctement :

```
gunicorn_django -b 0.0.0.0:8000
```

Votre application sera alors disponible sur le port 8000 de votre serveur de production. En cas de problème, la sortie console de Gunicorn vous indiquera l'erreur rencontrée. Il sera alors possible de la corriger et de poursuivre la mise en production, en créant un fichier de configuration dans le répertoire de configuration de Nginx (habituellement `/etc/nginx/conf.d/`). Voici à quoi doit ressembler votre configuration :

```
server {
    listen   80;
    server_name <example.net>;
   # no security problem here, since/is alway passed to upstream
    root <chemin complet vers la racine de votre projet>;
  # serve directly - analogous for static/staticfiles

    location /static/ {
        # this changes depending on your python version
        root <chemin complet vers la racine de votre projet>;
        expires max;
        add_header Pragma public;
        add_header Cache-Control "public, must-revalidate, proxy-revalidate";

    }
    location / {
        proxy_pass_header Server;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Scheme $scheme;
        proxy_connect_timeout 10;
        proxy_read_timeout 10;

        proxy_pass http://localhost:<un port de votre choix>/;
    }
   # what to serve if upstream is not available or crashes
    error_page 500 502 503 504 /static/50x.html;
}
```

Pour que cette configuration fonctionne, vous devez :

- remplacer `<exemple.net>` vers le nom de domaine de votre machine de production dans `settings.py` ;
- configurer `STATIC_URL` avec la valeur `/static/` ;
- configurer `STATIC_ROOT` avec la valeur `<chemin complet de votre projet>/static/` ;
- remplacer `<un port de votre choix>` par une valeur, par exemple `8000` ou `8001`. Notez bien cette valeur, car vous allez en avoir besoin dans un instant.

Cette configuration permet de servir les fichiers statiques de votre application via Nginx et le reste de votre application via Gunicorn.

Collectez l'ensemble des fichiers statiques dans le dossier `STATIC_ROOT` :

```
$ python manage.py collectstatic
```

Puis lancez Gunicorn avec le nom de port que vous avez sélectionné dans la configuration Nginx :

```
$ gunicorn_django -b localhost:<numéro de port>
```

Redémarrez votre serveur Nginx et testez votre application.

Afin que votre application puisse être lancée automatiquement en cas de redémarrage ou lorsque vous quittez le terminal de votre machine de production, utilisez Supervisor, un petit utilitaire qui va se charger de ces opérations de maintenance (http://supervisord.org/). Installez-le via le système de paquet de votre distribution ou via `pip` en tant que super-utilisateur.

Écrivez-lui ensuite un fichier de configuration, habituellement dans `/etc/supervisor/conf.d/` :

```
[program:agenda]
command= <chemin complet de votre environnement virtuel>/bin/gunicorn_django
<chemin complet de votre projet> -b localhost:<numéro de port>
directory=<chemin complet de votre projet>
user=<utilisateur qui lancera gunicorn>
autostart=true
autorestart=true
startsecs=10
redirect_stderr=true
stdout_logfile=/var/log/supervisor/agenda.gunicorn.log
```

Vous remarquerez que l'utilisateur `root` ne doit jamais lancer votre projet, et ce, pour des raisons de sécurité. En tant que super-utilisateur, lancez la commande de gestion :

```
$ supervisorctl
supervisor >
```

Vous pouvez avoir de l'aide sur cette interface de gestion avec `help` :

```
supervisor> help
default commands (type help <topic>):
=====================================
add    clear fg        open quit    remove restart   start   stop update
avail exit   maintail pid   reload reread shutdown status tail version
```

Ajoutez simplement votre projet :

```
supervisor> add agenda
```

et démarrez-le :

```
supervisor> start agenda
```

Vous devriez avoir maintenant une ligne de ce type dans la console de gestion de Supervisor :

```
supervisor> status
agenda                           RUNNING    pid 3697, uptime 19 days, 0:04:09
```

Votre application est maintenant disponible pour le monde entier. Cette configuration a l'avantage d'être particulièrement peu gourmande en termes de ressources.

OUTIL

Si vous souhaitez améliorer cette configuration selon vos besoins, en particulier en affinant la configuration de Gunicorn, vous trouverez toutes les ressources nécessaires sur le site du projet Gunicorn

- http://gunicorn.org

Conclusion

Votre application, si elle n'est pas encore parfaite, est désormais terminée. Vous avez maintenant tout le savoir nécessaire pour mener à bien les derniers ajustements. Néanmoins, nous ne vous abandonnons pas pour autant dans le développement de l'application. En effet, vous pourrez suivre les mises à jour du code sur https://bitbucket.org/boblefrag/livre_django_agenda. Au programme, nous reviendrons bien entendu sur les templates pour les rendre plus accueillants, comme nous l'avons fait au chapitre précédent. Nous tâcherons également de protéger l'ensemble des URL des applications par le décorateur `login_required`, comme dans l'application `usermanagement`. Vous constaterez également que nous factoriserons régulièrement le code et que nous ajouterons les docstrings qui manquent cruellement à ce projet.

En bref, même s'il reste encore du travail sur cette application et que par essence une application web n'est jamais terminée, vous avez tout le savoir pour la personnaliser. N'hésitez donc pas à laisser des commentaires sur le site Bitbucket au sujet des améliorations que vous avez trouvées, des problèmes que vous avez rencontrés et des solutions que vous y avez apportées.

C'est à vous de jouer maintenant !

10

Django et les bases de données

Ce qui fait qu'une application web est dynamique, c'est sa capacité à stocker de l'information et à la restituer sous diverses formes, selon différents critères. Les fonctionnalités d'enregistrement de données sont fournies par des logiciels appelés « bases de données » (ou SGBD, système de gestion de base de données). Django est en mesure de travailler avec un grand nombre d'entre elles. Vu que chacune possède des points forts et des points faibles, vous devez apprendre à les connaître afin de faire le bon choix au moment de la conception d'une application.

Tour d'horizon des bases de données relationnelles

La base de données est certainement l'élément le plus critique de votre application. De votre choix et de votre manière de l'implémenter dépendront en grande partie la robustesse et les performances de votre site. Il vous faudra donc choisir soigneusement le gestionnaire de bases de données (SGBD) que vous utiliserez, en fonction de vos besoins.

MySQL : SGBD historique

MySQL est certainement le SGBD historique du monde du logiciel libre. Profitant d'une grande communauté d'utilisateurs, il sera sans doute un bon choix si vous installez votre application sur un hébergement mutualisé.

Comme les bases de données MySQL sont employées depuis de nombreuses années, il existe beaucoup d'outils pour les manipuler, aussi bien en ligne de commande qu'en environnement graphique ou encore via une interface web.

PostgreSQL : robustesse

Le fonctionnement général de PostgreSQL est assez proche de celui de MySQL : mode client/serveur, implémentation du langage SQL… Ce SGBD concurrence de plus en plus MySQL dans les environnements professionnels, du fait principalement de sa robustesse.

Si vous souhaitez utiliser les modules géographiques de Django dans une application robuste, PostgreSQL sera le choix le plus logique.

SQLite : légèreté

SQLite est, comme son nom l'indique, un SGBD très léger. Il est également très simple à utiliser, ce qui en fait bien souvent l'outil de choix dans un environnement de développement.

Sa principale caractéristique, qui est à la fois son point fort et son point faible, est de produire des bases de données « fichiers ». Vous n'avez donc pas de système de gestion complexe pour manipuler votre base. De plus, les copies de sauvegarde et autres *backups* se bornent bien souvent à de simples copier-coller. Vous trouverez également très agréable de savoir qu'un connecteur SQLite est directement intégré au cœur de Python. Vous n'avez donc rien à installer pour en bénéficier dès maintenant. Les autres avantages de cette base de données minimaliste sont encore une fois liés à cette particularité.

Base de données en RAM

Comme SQLite construit des bases de données sous forme de fichiers, vous pouvez, si la persistance des données vous importe peu, le stocker directement en RAM et profiter des accès ultrarapides que cela vous offre. Bien entendu, lorsque votre ordinateur redémarre, votre base de données est totalement détruite. Vous aurez alors peut-être intérêt à utiliser `dumpdata` et `loaddata` pour charger et décharger les quelques données persistantes que votre base de données aura enregistrées.

Base de données géographique

Avec son module spatial, SQLite partage également avec PostgreSQL la capacité de gérer les modèles géographiques. Vous pourrez donc, sans la moindre difficulté, implémenter un système de gestion géographique performant avec ce SGBD minimaliste.

Application de bureau, application mobile

Si vous souhaitez créer une application mobile et/ou de bureau, SQLite est la solution idéale. Livrée par défaut sur les principaux terminaux mobiles, sans installation supplémentaire sur un bureau, elle vous offre un choix plus simple à employer qu'un fichier XML par exemple et, surtout, plus robuste.

Inconvénient majeur

SQLite a pourtant un inconvénient majeur : l'écriture concurrente. Puisque les bases de données sont sous forme de fichiers, plusieurs utilisateurs ne peuvent pas la modifier en même temps, car le système hôte impose en effet un blocage, qui ne dure que quelques millisecondes. Mais si votre application provoque un nombre conséquent d'écritures concurrentes, ce n'est certainement pas le bon choix à faire.

Comment choisir sa base de données ?

Bien choisir son gestionnaire de bases de données, c'est surtout une question de choix techniques. Il vous faut donc établir la liste de vos besoins.

- Quels sont les besoins de préservation des données ?
- Quelles sont les contraintes de performances ?
- Aurez-vous un gros volume de données ?
- Aurez-vous de nombreuses écritures concurrentes ?
- Aurez-vous besoin d'enregistrer et de manipuler des données spatiales ?
- Aurez-vous la possibilité de choisir votre gestionnaire de bases de données ?

Une fois que vous aurez répondu à ces questions, le choix vous semblera sans doute beaucoup plus clair. Pourtant, il se peut aussi que vous ne soyez pas encore vraiment satisfait. Par exemple, peut-être que la gestion purement « relationnelle » des outils précités ne vous convient pas ? Peut-être avez-vous besoin d'un système offrant de très fortes capacités de réplication ou de « scalabilité » ?

Heureusement, il vous reste encore une chance de trouver votre bonheur en vous tournant vers des outils spécialement conçus pour vous donner ce que vous cherchez : les SGBD NoSQL (*Not Only SQL*, « pas seulement SQL »).

Tour d'horizon des bases de données NoSQL

Les bases de données NoSQL sont très nombreuses et il est assez malaisé de les lister toutes. Cependant, les plus populaires sont sans doute MongoDB, Riak, CouchDB et Redis.

MongoDB

MongoDB est assez proche des gestionnaires de bases de données relationnelles. C'est un outil orienté document, c'est-à-dire que chaque enregistrement est vu comme un document dont le schéma peut être changeant. Autrement dit, vous n'avez pas à définir un schéma particulier pour vos données. C'est extrêmement souple d'utilisation.

Comme MongoDB supporte les relations, contrairement à bien des bases de données non relationnelles, vous pouvez l'utiliser comme base de données d'un projet Django, grâce à https://github.com/django-nonrel/django-nonrel.

Riak

Riak produit des bases de données clé/valeur, c'est-à-dire que chaque enregistrement est vu un peu comme un dictionnaire. Cette façon de faire autorise des enregistrements très différents les uns des autres, mais n'offre pas la gestion des relations.

Son gros point fort est sa capacité à grandir aisément et à être hébergé sur plusieurs machines de façon distribuée. Il est possible de l'intégrer dans Django. Par exemple, des projets se servent de Riak pour la gestion des sessions dans Django : https://github.com/flashingpumpkin/django-riak-sessions.

CouchDB

CouchDB vise le monde du Web. C'est une base de données orientée document, comme MongoDB, qui représente les éléments sous la forme d'objets JSON. Elle utilise le JavaScript pour la manipulation et le protocole HTTP comme API.

Le projet Couchdbkit (http://couchdbkit.org/) intègre CouchDB dans le cadre d'un projet Django. Il offre une interface d'accès assez similaire à l'ORM Django que vous employez habituellement.

Redis

Comme Riak, Redis est une base de données orientée clé/valeur. Chaque valeur peut être de type chaîne de caractères, liste, tableau, dictionnaire et dictionnaire ordonné.

Vous trouverez de nombreux projets Django qui l'emploient comme système de cache avec des performances tout à fait honorables, en utilisant par exemple django-redis (https://github.com/niwibe/django-redis.)

L'ORM de Django : couche d'abstraction entre la BDD et les objets

ORM signifie *Object Relational Mapper*. Il s'agit d'une couche d'abstraction entre votre base de données et les objets Python que vous manipulez. Si cet ORM simplifie beaucoup la gestion de votre base de données, comprenez bien que vous ne pourrez en faire bon usage que si vous possédez une bonne compréhension des notions sous-jacentes à l'ORM de Django. C'est ce que nous allons découvrir dans cette section.

Principe de fonctionnement

L'ORM de Django sert à abstraire la gestion de votre base de données. Il est dit « agnostique », c'est-à-dire qu'il fonctionne de manière identique quel que soit le SGBD retenu. C'est indéniablement un avantage puisque cela vous permet de développer votre application avec un SGBD (SQLite, par exemple) et de la mettre en production avec un autre. Cependant, il faut bien l'avouer, cela pose le problème du « plus petit dénominateur commun ». En effet, les points forts de certains SGBD ne sont pas disponibles avec Django car celui-ci doit rester compatible avec les autres, par exemple les *arrays* de PostgreSQL, qui ne sont pas reconnus par SQLite ou MySQL.

Vous échangez donc un peu de performance pour un peu plus de souplesse. Bien entendu, si vous le souhaitez, il est toujours possible d'implémenter, de votre propre chef, des méthodes particulières autorisant l'utilisation des fonctions spéciales de certains SGBD ; toutefois, c'est un peu ardu et cela vous fait bien entendu perdre en souplesse.

L'ORM de Django va donc implémenter un sous-ensemble de SQL et vous le rendre disponible sous forme d'objets Django aisément manipulables. Certaines fonctions qui ne sont pas disponibles dans tous les outils (les `DELETE ON CASCADE`, par exemple) seront simulées par une implémentation de Django, en Python donc.

Une vision objet

Au travers de l'ORM, Django vous propose donc de manipuler les résultats d'une requête en base de données comme un objet Python. La requête elle-même, la queryset, est traitée comme un objet. Cette vision de Django vous autorise donc à mélanger sans le moindre problème les objets purement Python et ceux des bases de données. Il est même tout à fait possible de créer des « pseudo-objets » de base de données, qui sont en fait des objets Python purs, et de les manipuler de la même manière, de les mélanger, d'en faire a peu près ce que vous voulez.

Les principaux avantages : manipuler les objets de la BDD en Python

Le principal avantage de l'ORM de Django est sans nul doute le fait de manipuler les objets de la base de données en Python. En effet, cela vous évite d'apprendre le SQL, même si la connaissance de fortes notions est toujours conseillée, et facilite également l'interfaçage du code Django et des objets de la base de données. C'est, par exemple, ce que vous avez fait lorsque vous avez créé des méthodes au sein de vos modèles. Ces derniers sont la représentation Python de la structure de votre base de données et les méthodes les manipulent directement en Python.

L'implémentation purement Python de l'ORM de Django offre de nombreux avantages pratiques, comme les notions d'héritage qui n'existent pas dans les bases de données relationnelles (exception faite de PostgreSQL) ; ces dernières rendent le code plus concis, l'application plus portable et elles donnent une très grande souplesse dans la gestion des données, avec les héritages « réels », « abstraits » et « proxy ».

Django a également prévu de nombreux points d'entrée pour adapter l'ORM un peu comme bon vous semble. Dans le chapitre 16 « Django Avancé », nous analyserons certaines de ces techniques, comme les querysets héritées qui tirent pleinement parti de l'héritage « réel ».

Les inconvénients : comment les contourner

Les principaux inconvénients de l'ORM de Django viennent de ses qualités. En effet, comme il est agnostique, c'est-à-dire qu'il se limite au plus petit dénominateur commun aux divers SGBD, il ne vous permet pas de bénéficier pleinement des capacités étendues de votre outil préféré.

Comme l'ORM de Django vous évite d'écrire directement du SQL, le code rédigé en Python peut se révéler désastreux en termes de performances sur la base de données. De même, les capacités d'héritage de l'ORM sont extrêmement pratiques, mais elles risquent aussi de poser de gros problèmes de performances si l'on n'y prend pas garde.

Voici donc quelques conseils pour ne pas tomber dans le piège de la facilité et faire chuter inconsidérément les performances.

Anatomie d'une queryset

Comme le premier écueil auquel vous allez être confronté concerne la queryset, observons-la de plus près.

La première chose à savoir est qu'elle n'est exécutée qu'au besoin. Django va en effet attendre le tout dernier moment pour effectuer réellement la requête en base de données. Si, par exemple, vous écrivez :

```
requete = MonModel.objects.filter(name = 'hello')
return 'ceci est un test'
```

votre requête ne sera tout simplement pas exécutée. Django va la garder en mémoire et ne l'exécutera que lorsque ce sera nécessaire. De cette fonctionnalité découlent plusieurs choses.

Vous pouvez filtrer en plusieurs fois votre queryset ; son évaluation ne sera faite qu'une seule fois. Par exemple :

```
hommes = Auteur.objects.filter(sexe = 'homme')
celibataires = hommes.filter(situation = 'celibataire')
```

est parfaitement égal à :

```
celibataires = Auteur.objects.filter(
              sexe='homme',situation = 'celibataire')
```

Une boucle `for` sur une queryset va exécuter cette dernière et charger l'ensemble des objets de votre requête en mémoire. Cela peut être absolument désastreux pour votre machine. Prenons un exemple où la base de données contient un million d'articles :

```
articles = Article.objects.all()
for art in articles :
# Vous venez de charger en mémoire 1 million d'articles. Il est probable que
votre machine ne le supporte pas. Voyons comment contourner le problème.
articles = Articles.objects.all()
max = articles.count()
# Ici nous faisons un « count » qui est donc évalué par la base de données.
C'est à la fois beaucoup moins violent pour la base de données, mais également
pour la mémoire puisque l'on ne charge que le nombre d'objets disponibles.
slice = 1000
# Ici, nous découpons la requête en « paquets » de 1 000 objets de manière à ce
qu'à un instant t ne soient chargés en mémoire que 1 000 objets au maximum.
end = 0
while end < max :
    start = end
    end += slice
    for art in articles[start:end] :
       # Ici, il n'y a plus que 1 000 objets en mémoire.
```

Avec un tel code, n'oubliez pas que vous faites tout de même 1 000 requêtes dans la base de données. S'il s'agit d'une opération de maintenance, cela peut être tout à fait convenable, mais pour afficher des résultats à l'utilisateur, vous auriez tout intérêt à revoir votre code ou à filtrer votre requête pour n'avoir qu'un sous-ensemble des résultats par exemple.

Méfiez-vous des templates

Le rôle du template est d'afficher les résultats d'un calcul que vous avez effectué dans votre vue. Cependant, la propriété « juste-à-temps » des querysets peut vous jouer des tours. En voici quelques exemples et comment les contourner.

Maintenant que vous savez que les requêtes ne sont exécutées qu'au besoin, vous comprendrez sans doute l'intérêt de renvoyer une queryset au template et non pas une liste d'objets. Renvoyer une queryset, donc un *callable*, permet de ne pas exécuter la requête en base de données si cela n'est pas requis. Cependant, ne prenez pas cette règle au pied de la lettre. La aussi, cela doit être pensé au cas par cas. Par exemple, si votre template se présente comme suit :

```
{% for article in articles %}
{{article.titre}}
{%endfor%}
{% for article in articles %}
{% article.date %}
{%endfor%}
```

dans le cas où `articles` est une liste d'objets, la requête a déjà été exécutée au niveau de votre vue. Une seule requête sera donc effectuée dans la base de données. En revanche, si `articles` est un callable (une queryset donc), votre requête sera exécutée pour chaque boucle (ici, deux fois). Pour résoudre ce problème, utilisez `with` : ce template tag va enregistrer le résultat d'une requête dans votre template pour une utilisation récurrente :

```
{% with articles as arts %}
{% for article in arts %}
{{article.titre}}
{%endfor%}
{% for article in arts %}
{% article.date %}
{%endfor%}
```

Cette fois, l'utilisation de `with` permet d'enregistrer le résultat de la requête et c'est donc ce dernier qui est utilisé tout au long de votre template. La base de données n'est donc sollicitée qu'une seule fois.

Comme dans un programme Python classique, Django vous invite à faire « la chasse aux points ». Si les gains en performances d'un programme Python sont généralement mineurs, dans le cas des templates, c'est beaucoup plus important. Pour vous en convaincre, prenons l'exemple suivant :

```
{% for usr in user.profile.circle.users_set.all %}
{{usr.profile.circle}}
{%endfor %}
```

Ce code, a priori sans danger, est en réalité une bombe thermonucléaire pour les performances de votre application.

Vous avez une liste d'objets `user`. Pour chacun, récupérez le cercle qui lui correspond (une recherche sur la clé étrangère de l'objet `circle`), puis chargez tous les utilisateurs liés à ce cercle, via une autre clé étrangère donc. Ensuite, dans votre boucle, itérez sur chaque utilisateur et chargez, via une clé étrangère, le cercle auquel il appartient.

Si réellement vous avez besoin d'effectuer ce type de recherche et si vous savez qu'il vous faudra atteindre le cercle des utilisateurs liés à un utilisateur défini, nous vous conseillons fortement de relier directement les cercles et l'utilisateur, de manière à écrire :

```
{% for circle in user.profile.users_circle %}
{{ circle }}
{% endfor %}
```

Les différents types de champs

Les différents types de champs à votre disposition dans l'ORM correspondent à ceux que l'on trouve habituellement dans une base de données : les champs de type chaîne de caractères, entier ou booléen. Pourtant, comme la logique de validation des champs est liée très logiquement aux modèles, vous trouverez aussi des champs de type `email` qui ne sont rien d'autre que des champs de type chaîne de caractères dont la validation vérifiera l'e-mail.

Les champs CharField

Les champs de type `CharField` sont les plus simples que vous pourrez trouver. Ils correspondent à une chaîne de caractères dont la longueur maximale est obligatoirement définie. Django valide la longueur de la chaîne de caractères et affiche à l'utilisateur un champ de saisie de type texte.

Les champs IntegerField

Les champs de type `IntegerField` sont très similaires aux précédents. Comme eux, ils ont une valeur maximale obligatoire, ainsi qu'une valeur minimale, et Django valide le respect de cet intervalle. Il vérifie également que les données saisies sont bien de type entier et présente à l'utilisateur un champ de saisie de type texte.

Les champs TextField

Les champs de type `TextField` vous permettent d'entrer une grande quantité de texte. Il n'y a donc pas de longueur maximale ni de validation. Django présente un champ de type `textarea` dans les formulaires.

Les options remarquables

Tous les champs de modèles partagent des options communes.

Options relatives aux formulaires

Cet ensemble d'options contrôle la manière dont le champ est affiché à l'utilisateur.

- `default` indique une valeur par défaut si le champ est vide.
- `verbose_name` : habituellement, Django emploie le nom que vous donnez au champ pour le présenter à l'utilisateur dans un formulaire. Parfois, vous préférerez un nom plus parlant ou internationalisé, que vous spécifierez dans `verbose_name`.
- `verbose_name_plural` précise un pluriel irrégulier (cheval/chevaux, par exemple) pour le nom de champ.
- `editable` : `False` ou `True` (valeur par défaut). Si vous spécifiez la valeur `False`, ce champ ne sera pas éditable, mais calculé (un champ `total` dans une addition, par exemple).
- `help_text` : cette option définit un texte qui sera affiché à côté du champ pour aider l'utilisateur à le remplir.

Options relatives à la base de données

Cet ensemble contrôle la manière dont un champ est enregistré en base de données :

- `blank`
- `null`
- `db_column`
- `db_index`
- `db_tablespace`

- `primary_key`
- `unique`

Les autres types de champs

De nombreux autres types de champs sont disponibles dans Django. Vous en trouverez la liste complète sur le site officiel de la documentation. Plutôt que de vous proposer une liste fastidieuse des champs proposés par Django et de leurs différentes options, apprenons à créer nos propres types de champs selon nos besoins.

Créez vos propres types de champs

Pour bien comprendre comment fonctionne la logique de Django sur ce point, il est nécessaire de faire un petit retour en arrière. Lorsque vous créez vos modèles, vous les rédigez comme ceci :

```
from django.db import models
class MyModel(models.Model) :
    name = models.CharField(max_length= 250)
```

Ici, vous venez de définir une classe héritant de `models.Model` dont l'attribut `name` est de type `CharField`. Vraiment ? Regardons de plus près : `CharField` est en réalité une classe. `name` est donc une instance de la classe `CharField` dont la longueur maximale a été définie à 250 caractères.

Comprendre que les attributs des classes modèles sont des classes aide à saisir plus rapidement comment vous allez pouvoir créer vos propres types de champs. Prenons un exemple très simple : imaginons que vous souhaitiez que tous vos champs de type `CharField` aient une longueur maximale de 250 caractères. Vous écrirez :

```
class CustomCharField(models.CharField):

    def __init__(self, *args, **kwargs):
        max_length = 250
        super(CustomCharField, self).__init__(
            *args,
             max_length=max_length,
             **kwargs)


class MyModel(models.Model):
    name = CustomCharField()
```

La fonction `__init__` est appelée lorsque vous utilisez `syncdb`, au moment de la création des tables donc. Ici, nous n'indiquons qu'à un seul endroit clairement identifiable la valeur de `max_length` pour tous les champs de type `CharFieldCustom`. L'usage de `super` permet alors d'initialiser normalement votre champ. Pour Django, cela reste un champ de type `CharField` comme les autres.

Cet exemple reste bien entendu très simple ; il n'a pour but que de vous montrer la logique de fonctionnement de Django. Arrêtons-nous néanmoins sur deux méthodes remarquables que vous serez amené à utiliser si vous souhaitez surclasser les types de champs.

Dès que vous voudrez réaliser des actions un peu plus complexes sur vos champs, vous aurez besoin de deux autres méthodes : `to_python` et `formfield`. Elles ne sont pas implémentées pour tous les champs, mais seulement pour ceux qui le nécessitent. Le fonctionnement de ces deux méthodes est simple : dès que la valeur que vous devez afficher à l'utilisateur doit être modifiée pour être compatible avec un type Python, vous devez utiliser `to_python`. Si vous souhaitez modifier le champ de formulaire qui sera affiché à l'utilisateur, ce sera `formfield`.

Pour mieux comprendre comment fonctionnent ces deux valeurs, prenons en exemple le code de Django lui-même et en particulier le champ `DateField`. Il n'existe pas, dans une base de données SQLite par exemple, de champ permettant de définir une date. Django est donc obligé d'enregistrer une chaîne de caractères au niveau de la base de données, bien qu'il manipule un objet date avant et après l'enregistrement. Voyons à quoi ressemble la méthode `to_python` de ce champ :

```
    def to_python(self, value):
        if value is None:
            return value
        if isinstance(value, datetime.datetime):
            return value.date()
        if isinstance(value, datetime.date):
            return value

        value = smart_str(value)

        try:
            parsed = parse_date(value)
            if parsed is not None:
                return parsed
        except ValueError:
            msg = self.error_messages['invalid_date'] % value
            raise exceptions.ValidationError(msg)

        msg = self.error_messages['invalid'] % value
        raise exceptions.ValidationError(msg)
```

Le code est assez simple. S'il ne parvient pas à déterminer le type de la valeur (soit `None`, soit un objet `datetime`, soit un objet `date`), la valeur est convertie en chaîne de caractères (avec la fonction `smart_str`), puis Django effectue la validation du champ. Dans le cas de `value_to_string`, c'est exactement l'inverse qui est effectué. Django récupère la valeur en base de données et la présente à l'utilisateur.

En ce qui concerne `formfield`, c'est même plus simple encore. Reprenons le code du champ `DateTime`. Le champ `DateField` hérite de `Field`, qui est la classe de base pour de nombreux champs. `Field` implémente déjà une méthode `formfield` et présente à l'utilisateur un champ de formulaire de type texte. Ici, on veut un champ de type date, tout en conservant le fonctionnement initial et assez complexe de cette méthode. Voici comment Django va modifier le widget pour ce type de champ :

```
def formfield(self, **kwargs):
    defaults = {'form_class': forms.DateField}
    defaults.update(kwargs)
    return super(DateField, self).formfield(**defaults)
```

Django a seulement modifié le dictionnaire `default` en changeant la valeur de `form_class`. C'est une méthode rapide et simple pour modifier le type de champ affiché à l'utilisateur. Lorsque nous aborderons les formulaires, vous verrez qu'il existe d'autres méthodes pour modifier les types de champs dans les formulaires. N'oubliez pourtant pas celle-ci car c'est dans certains cas une solution plus simple et moins verbeuse.

Les relations

L'un des principaux avantages des bases de données relationnelles est la possibilité de créer des liens entre différentes tables. C'est d'ailleurs pourquoi on dit qu'elles sont relationnelles. Django propose donc, via son ORM, des méthodes simples pour gérer ce type de relation.

Les relations d'une base de données peuvent être de plusieurs types.

- Le premier cas d'utilisation concerne les relations de type 1 vers 1 : un enregistrement dans la base de données de type A est lié à un seul autre enregistrement de type B, qui lui-même ne peut être lié qu'à un seul enregistrement de type A. Par exemple, un profil est lié à un utilisateur et un utilisateur ne peut avoir qu'un seul profil. Cette relation est dite de type *one-to-one*. Elle peut être insérée indifféremment sur l'un ou l'autre des types.
- Le deuxième cas d'utilisation concerne la relation 1 vers N. C'est un type de relation où un enregistrement de type A est lié à un ou plusieurs enregistrements de type B. Ces derniers sont, eux, liés à un seul enregistrement de type A. Par exem-

ple, un utilisateur peut être lié à plusieurs messages, mais chaque message n'émane que d'un seul utilisateur. Il s'agit d'une relation de type *foreign key* (ou clé étrangère), puisque sur le type d'enregistrement B, on insère une référence à un type d'enregistrement différent.

- Enfin, le dernier type est la relation N vers N. Il s'agit ici de lier entre eux deux types de modèles sans tenir compte du nombre de relations qu'ils partagent. Ainsi, l'enregistrement de type A peut être lié à plusieurs enregistrements de type B qui eux-mêmes peuvent être liés à plusieurs enregistrements de type A. Par exemple, un utilisateur appartient à plusieurs groupes, qui eux-mêmes comprennent plusieurs utilisateurs. Il s'agit de relations *many-to-many*.

Django vous offre donc la possibilité de spécifier ces types de relations dans les modèles. Il permet aussi, et c'est sans doute le plus important, de traverser les relations. Par exemple, grâce à l'ORM, vous serez en mesure de répondre aux questions suivantes.

- Quel est le profil de l'utilisateur A ?
- Quels sont les messages de l'utilisateur A ?
- Quels sont les groupes de l'utilisateur A ?

Et vous pourrez également répondre aux questions inverses.

- À qui appartient ce profil ?
- Qui est l'auteur de ce message ?
- Quels sont les utilisateurs qui sont liés à ce groupe ?

Les relations one-to-one

Une relation 1 vers 1 correspond en quelque sorte à une relation de type foreign key, ou relation 1 vers N, avec une contrainte d'unicité. Cela revient à dire dans le langage courant : « relation de 1 vers plusieurs où plusieurs ne peut être supérieur à 1 ».

La principale différence entre les deux approches se trouve au niveau de l'ORM. Quand vous souhaiterez faire une requête en partant du modèle référencé, dans un cas vous obtiendrez une liste de résultats ne contenant qu'un objet, dans l'autre cas, ce sera directement l'objet.

Cas d'utilisation

Le cas le plus courant de l'utilisation d'une relation one-to-one concerne l'extension d'un autre modèle. C'est extrêmement pratique quand vous ne souhaitez pas modifier un modèle existant, mais que vous voulez malgré tout lui ajouter de nouveaux champs. Django utilise d'ailleurs cette façon de faire avec l'héritage de table multiple.

Implémentation

Pour insérer une relation 1 vers 1, prenez l'un des deux modèles que vous souhaitez lier entre eux et ajoutez un champ `OneToOneField` référençant l'autre modèle.

Dans l'exemple qui consiste à créer un profil pour chaque utilisateur, vous pourriez écrire :

```
from django.contrib.auth.models import User
from django.db import models

class Profile(models.Model) :
    user = OneToOneField(User)
```

Queryset sur une relation one-to-one

Comme une relation one-to-one ne renvoie qu'un objet, vous pouvez vous permettre de l'employer comme si elle était un champ du modèle. Par exemple, pour obtenir l'utilisateur qui a écrit un commentaire, écrivez :

```
class Commentaire(models.Model) :
    user = ForeignKey(User)

>>> commentaire.user.username
```

Dans le sens inverse, la même syntaxe est employée, par exemple pour obtenir le nom de l'utilisateur :

```
>>> user.profile.commentaire
```

Si vous le souhaitez, et cela est vrai pour tous les types de relations, utilisez un argument spécial lors de la création de votre modèle `related_name`. Il sert à modifier le nom par lequel sera effectuée la relation dans le sens inverse. Si vous ne précisez pas de champ `related_name`, Django utilisera par défaut le nom du modèle, en minuscules :

```
>>> user.profile
```

Si, en revanche, vous avez précisé un champ `related_name`, par exemple :

```
class Profile(models.Model) :
    user = OneToOneField(User, related_name='details')
```

vous devrez utiliser la syntaxe suivante pour atteindre le modèle `Profile` :

```
>>> user.details
```

Les foreign keys (clés étrangères)

Les foreign keys, ou clés étrangères, servent à modéliser les relations de type 1 vers N. Elles représentent souvent des relations du type appartenance, comme « Quelles sont les voitures produites par cette marque ? » Elle fonctionnent bien entendu dans le sens inverse et répondent donc à la question « Quelle est la marque qui produit cette voiture ? »

Le danger des relations de type clés étrangères est que vous ne maîtrisez pas le nombre d'objets que peut vous retourner la base de données. Imposez-vous donc une limite lors de l'évaluation de vos querysets ou utilisez un système de pagination. Faites également très attention à vos *selects* dans vos formulaires sur des relations de type foreign key ou many-to-many.

Implémentation

Les clés étrangères doivent être placées sur l'objet pour lequel il existe une référence à un objet extérieur. Ainsi, dans notre exemple sur les marques de voitures, c'est sur l'objet `voiture` que se trouvera la référence à l'objet `marque`. Une relation de type clé étrangère se définit comme suit :

```
class Voiture(models.Model) :
    marque = models.ForeignKey(Marque)
```

Queryset sur une relation foreign key

Comme une relation de type clé étrangère ne peut renvoyer qu'un seul objet, la syntaxe est identique à celle de la relation one-to-one, tout du moins dans le sens « objet qui référence » vers « objet référencé ». Ainsi, pour retrouver la marque d'une voiture, vous pouvez écrire :

```
>>> voiture.marque.nom
```

En revanche, si vous souhaitez inverser le sens de la relation et donc retrouver les voitures conçues par la marque en question, il faut écrire :

```
>>> marque.voiture_set.all()
```

Comme vous pouvez le remarquer, vous n'avez plus affaire à un champ, mais à une méthode. Cette dernière, que vous n'avez pas eu besoin d'écrire, est un manager. La principale différence avec les derniers exemples, c'est qu'ici vous n'avez pas un seul objet mais une collection. Comme il est probable que vous ne souhaitiez pas les traiter tous, et qu'en tout cas vous ne voulez pas faire inconsidérément une requête en

base de données qui peut s'avérer coûteuse, Django crée pour vous un générateur de querysets, qui sont ce que l'on appelle des « managers » (voir la section « Les managers », page 242). Pour le moment, sachez que leur utilisation est identique à celle d'une queryset classique. En effet, quand vous utilisez la syntaxe qui suit, votre manager n'est autre que `objects` :

```
>>> marque.objects.all()
```

`voiture_set.all()` n'est pas fondamentalement différent, à ceci près que le manager a déjà filtré pour vous les voitures qui ont une clé étrangère vers la marque que vous manipulez. Ainsi, les deux écritures suivantes sont équivalentes :

```
>>> voiture.objects.filter(marque=marque)
>>> marque.voiture_set.all()
```

Comme vous avez affaire à une queryset, manipulez-la comme bon vous semble, en écrivant par exemple :

```
>>> marque.voiture_set.filter(couleur='rouge')
```

related_name

Comme pour les relations one-to-one, vous pouvez préciser un `related_name` dans votre modèle. Par défaut, Django nomme la relation inverse par le nom du modèle de référence en minuscules suivi de `_set`. Si vous précisez un `related_name`, par exemple :

```
class Voiture(models.Model) :
    marque = ForeignKey(Marque, related_name='voitures')
```

vous pouvez ensuite utiliser la syntaxe suivante pour retrouver les voitures conçues par un fabriquant :

```
>>> marque.voitures.all()
```

Modéliser des relations hiérarchiques

Les relations de type clés étrangères servent également à modéliser des relations hiérarchiques. Imaginons que vous souhaitiez implémenter dans un moteur de blog un système de catégories et de sous-catégories sur plusieurs niveaux. Il est possible d'obtenir la fonctionnalité escomptée en utilisant une référence à un modèle du même type.

Par exemple :

```
class Categorie(models.Model) :
    titre = models.CharField(max_length=150)
    parent = models.ForeignKey(
        Categorie,
        null=True
    related_name='enfants')
```

Nous avons précisé `null=True`, car une catégorie peut ne pas avoir de parent s'il s'agit de la catégorie « racine ». Cette relation retrouve le parent et les enfants d'une catégorie donnée :

```
>>> categorie.parent
>>> categorie.enfants.all()
```

Les relations many-to-many

Les relations many-to-many sont des relations N vers N. Elles sont utilisées quand vous avez besoin de modéliser des références croisées. Imaginons que vous souhaitiez modéliser un objet facture. Chaque facture contiendra un ou plusieurs produits et un produit sera présent sur une ou plusieurs factures. Ce type de modélisation répond aux questions : « Quels sont les produits présents sur cette facture ? » et « Quelles sont les factures sur lesquelles apparaît ce produit ? »

Implémentation

L'implémentation d'une relation many-to-many se fait sur l'un ou l'autre des modèles :

```
class Facture(models.Model) :
    produits = models.many-to-manyField(Produit)
```

Queryset sur une relation many-to-many

Dans le cas d'une relation many-to-many, comme vous manipulez toujours une collection d'objets, Django vous fournit toujours une queryset. C'est donc assez similaire à une relation de type foreign key inversée mais, cette fois, dans les deux sens ! Dans l'exemple proposé, vous pourrez retrouver tous les produits d'une facture en écrivant :

```
>>> facture.produits.all()
```

ce qui, comme vous l'avez déjà expérimenté avec les clés étrangères, vous donnera une queryset que vous pourrez évaluer ou filtrer selon vos besoins.

L'autre sens de la relation suit une nouvelle fois le même principe que les relations de type foreign key, avec le même système d'écriture. Ainsi, pour retrouver les factures sur lesquelles apparaissent un produit, écrivez :

```
>>> produit.facture_set.all()
```

Là encore, vous manipulerez une queryset.

related_name

Comme pour les autres relations, simplifiez vos écritures en précisant un `related_name` qui sera employé pour les relations inversées. Par exemple :

```
class Facture(models.Model) :
    produits = models.many-to-manyField(Produit, related_name ='factures')

>>> facture.produit.all()
>>> produit.factures.all()
```

Manipulation des relations many-to-many

Contrairement aux deux précédents types de relations, où vous aviez toujours un objet unique sur lequel placer la relation, cette fois vous avez de multiples éléments des deux côtés. Se pose donc le problème de l'enregistrement de la relation.

Django vous offre une série de méthodes pour manipuler les relations many-to-many. Tout d'abord, si vous souhaitez ajouter une relation, par exemple un produit à une facture, vous utiliserez `add` comme suit – `un_produit` étant un objet de type `Produit` :

```
>>> facture.produit.add(un_produit)
```

Lorsque vous souhaitez supprimer un élément de la relation, par exemple un produit d'une facture, utilisez `remove` :

```
>>> facture.produit.remove(un_produit)
```

Enfin, pour supprimer l'ensemble des relations, employez `clear`. Ainsi, pour supprimer tous les produits d'une facture, écrivez :

```
>>> facture.produit.clear()
```

Ces trois actions (`add`, `remove` et `clear`) peuvent être effectuées dans l'autre sens. Ainsi :

```
>>> produit.factures.add(une_facture)
>>> produit.factures.remove(une_facture)
>>> produit.facture.clear()
```

sont des actions parfaitement valides.

Prenez garde cependant : ces trois actions ont un effet immédiat sitôt qu'elles sont appelées. En effet, contrairement aux modifications des autres champs d'un modèle, qui ne sont prises en compte qu'au moment de l'appel de la méthode `save`, ces actions sont effectuées immédiatement.

Les héritages de tables

Les héritages de tables dans Django servent à mettre en place des fonctionnalités très puissantes ou, tout simplement, à vous simplifier grandement l'écriture du code. Django propose plusieurs types d'héritages. Choisissez soigneusement celui qui vous convient le mieux, car il est assez difficile de revenir en arrière une fois que vous avez fait votre choix.

Héritage abstract

Le type d'héritage `abstract` n'a aucune incidence sur votre base de données. En revanche, il rend de grands services pour la concision de votre code. Il est également très couramment employé dans le cadre des applications réutilisables.

Le principe d'une classe abstraite dans les modèles consiste donc à définir un ensemble de champs et de méthodes, dans une classe dont ses enfants bénéficieront. Ainsi, vous n'écrivez qu'à un seul endroit les méthodes ou les champs que partagent nombre de vos modèles. Un seul endroit, clairement identifiable ? Vous l'aurez compris, une classe abstraite vous permet de rester DRY dans vos modèles.

Vous définissez votre classe abstraite comme n'importe qu'elle autre, par exemple :

```
class Header(models.Model) :
    '''cette classe permet de définir les champs communs aux médias '''
    titre = models.CharField(max_length=250)
```

Cette classe est volontairement très succincte. Vous avez donc défini une classe héritant de `models.Model` et dont l'unique attribut est un champ `titre`. Pour que cette classe devienne abstraite, ajoutez simplement une *inner class* `Meta` :

```
class Meta :
    abstract = True
```

Vous pouvez ensuite hériter de cette classe comme ceci :

```
class Article(Header) :
    contenu = models.TextField()
```

Votre classe `Article` possède maintenant un champ `titre` dont elle hérite par `Header` et un champ `contenu` que vous venez de définir.

Cette technique ne vous offre rien d'autre qu'une simplicité d'écriture. Vous ne pouvez pas, par exemple, faire des requêtes ou quoi que ce soit d'autre sur la classe `Header`, car elle n'existe que pour vous faciliter l'écriture de vos classes. Si vous souhaitez en faire plus, vous aurez besoin d'utiliser l'héritage multitable.

Les classes abstraites et les applications réutilisables

L'une des utilisations les plus classiques de l'héritage abstrait concerne la réutilisation d'applications. Lorsque vous créez une application qui a vocation à être distribuée sur plusieurs projets, vous définissez un ensemble de modèles contenant les champs et méthodes qui vous semblent utiles pour votre application. Il est fort probable que pour un autre projet, vous sentiez la nécessité de définir de nouveaux champs pour les modèles de l'application. Si ces modèles sont concrets, cela peut sembler un peu compliqué. Le plus simple, dans ce cas, c'est de mettre en pratique le principe des « applications réutilisables » comme il a été défini dans une conférence restée célèbre : « *Why Django sucks and how we can fix it?* » (en français, cela se traduirait par « Pourquoi Django est-il mauvais et comment peut-on arranger les choses ? »).

Le concept d'applications réutilisables va donc consister à créer des modèles « abstraits » ; les utilisateurs de votre application s'en serviront pour en hériter, dans leurs propres modèles. Ils bénéficieront alors, en plus des champs et méthodes qu'ils auront eux-mêmes définis, des champs et méthodes que vous aurez déterminés pour votre application. Cela offre donc à l'utilisateur une véritable souplesse d'usage. Si vous avez suffisamment découpé votre application, il pourra, par exemple, n'en utiliser qu'une partie, laissant de côté celles dont il n'a pas l'utilité.

Multitable

L'héritage multitable que propose Django se comporte comme un système d'héritage classique. Cette fois, contrairement à l'héritage abstrait, l'ORM va créer différentes tables suivant l'architecture que vous allez définir. Dans ce cas, les classes « mères » sont de vraies tables, reliées à leurs « filles » par des relations SQL.

Un contexte d'utilisation classique de l'héritage multitable concerne, par exemple, les documents multiformes. Imaginons une application où vous gérez plusieurs types de médias (photo, vidéo, audio, par exemple). Vous pourrez enregistrer l'auteur et la date de création de chaque fichier dans une classe `Media` et définir des champs spécifiques pour les `Photos`, `Vidéos` et `Sons`. Il vous sera ainsi possible de récupérer tous les médias d'un auteur spécifique ou créés à une date donnée.

Proxy

L'héritage de type proxy est un peu particulier. D'ailleurs, on ne peut pas vraiment parler d'héritage, mais plutôt de « substitution ». En effet, l'idée des proxy est de créer une copie d'un modèle existant et de lui ajouter des méthodes spécifiques. Ce type d'héritage ne supporte pas l'ajout de nouveaux champs ni ne permet le multi-héritage. En effet, si pour Django il y a bien plusieurs classes différentes, au niveau de la base de données en revanche, il n'y a qu'une seule table.

Un usage courant des proxy concerne l'interface d'administration. Si, par exemple, vous souhaitez proposer l'édition d'articles avec des droits différents suivant les utilisateurs, vous pouvez réaliser une classe `Article` dont les statuts sont « publié », « brouillon » et « soumis à publication ». L'administrateur aura accès à cette classe, et il pourra éditer et publier des articles. En revanche, via le système de droits, vous créerez un autre groupe, `Auteurs`, qui n'aura accès qu'à un sous-ensemble d'articles (ceux dont le statut de publication est à « brouillon ») qui ne pourront prendre que deux états : « brouillon » et « soumis à publication ». L'administrateur du site pourra ainsi relire les articles soumis à publication et ne publier que ceux qu'il jugera dignes de l'être.

Comme vous pouvez le constater, il y aura toujours une méthode d'héritage qui correspondra à vos besoins. L'important est de bien choisir celle qui vous correspond le mieux.

Les méthodes des modèles

Nous avons déjà abordé les méthodes des modèles lors des différents tutoriels. Ici, nous vous proposons d'aller un peu plus loin dans l'utilisation de ces méthodes.

Quelques remarques : gérer les appels à la base de données

Lorsque vous commencez vos premières applications Django, vous avez tendance à n'utiliser que très peu les méthodes des modèles. En revanche, vos fichiers `views.py` deviennent rapidement très volumineux. La raison est simple : la première idée qui vient est de créer toutes ses requêtes dans les vues. Par exemple, pour rendre au template l'ensemble des articles d'un auteur, vous allez écrire dans votre vue :

```
author = Author.objects.get(pk=1)
articles = Article.objects.filter(author=author)
```

On pourrait arguer que l'écriture qui suit est plus succincte :

```
articles = Author.objects.get(pk=1).articles_set.all()
```

Néanmoins, vous vous apercevrez qu'il existe une façon de faire à la fois plus simple et plus élégante, mais également plus lisible.

Dans le cas où vous faites régulièrement appel à cette queryset, il semble judicieux de pouvoir l'enregistrer quelque part. Or, y a-t-il meilleur endroit que le modèle lui-même ? Dans votre fichier `models`, vous allez donc écrire :

```
def articles(self) :
    return self.articles_set.all()
```

Et dans votre vue, à chaque fois que vous devrez récupérer les articles d'un modèle, vous n'aurez plus qu'à écrire :

```
articles = Author.objects.get(pk=1).articles()
```

Bien entendu, dans l'exemple courant, cela ne change pas grand-chose, mais continuons la réflexion un peu plus loin.

Au fil de votre développement, vous vous apercevez que, finalement, récupérer tous les articles d'un auteur est bien trop coûteux. Vous voudriez désormais récupérer les articles datant de moins d'une semaine. Si vous avez écrit la queryset dans toutes vos vues, un gros travail vous attend, puisque vous devez modifier chacune des querysets que vous avez écrites. En revanche, si vous avez eu la bonne idée de placer cette queryset comme une méthode de votre modèle, vous n'avez qu'une ligne à changer. Au fil du temps, vous ferez une liste des querysets que vous utilisez fréquemment dans les méthodes de vos modèles. Vous pourrez alors les optimiser à votre guise, les modifier pour qu'elles correspondent plus précisément à vos besoins…

En conclusion, nous vous conseillons fortement de gérer dans vos modèles tout ce qui concerne les appels à la base de données. Les vues ne doivent être là que pour récupérer des données utilisateur, appeler des méthodes des modèles, choisir un template et retourner l'ensemble à l'utilisateur.

Les meta

Les meta dans les modèles sont un moyen très simple de définir des « options » qui concernent le modèle lui-même et non pas l'un de ses champs. Par exemple, quand vous définissez un champ de modèle avec `verbose_name = 'titre'`, vous vous attendez à ce que ce champ soit affiché avec le label « titre » dans un formulaire. La classe `Meta` fonctionne exactement de la même manière. Par exemple, il est possible de définir l'option `verbose_name` pour que le nom du modèle s'affiche dans l'`admin` comme vous le souhaitez.

Lorsque nous avons parlé des héritages de modèles, nous avons utilisé la classe `Meta` pour définir le type d'héritage : `abstract`, `proxy`... Vous pouvez également définir de nombreuses autres choses.

Les échanges avec la base de données

Django présume bien des choses à propos de la base de données, ce qui vous conviendra la plupart du temps. Pourtant, dans certains cas, cela peut sembler être un frein. Imaginons, par exemple, que votre base de données soit préexistante ; il y a de grandes chances pour que le nommage des tables ne corresponde pas à ce que Django définit naturellement. Il écrit le nom des tables selon le gabarit suivant : `nom de l'application_nom du modèle`. Par exemple, le modèle `article` de l'application `blog` sera représenté au niveau de la base de données par `blog_article`. Si la base de données est préexistante à l'application, vous devez indiquer à Django le nom de la table, avec l'option `db_table`.

Dans le cas où la base de données existait avant que l'application ne soit créée, vous ne souhaitez peut-être pas que Django crée pour vous une nouvelle table lors du `syncdb` et la supprime lors du `reset`. Dans ce cas, utilisez l'option `managed=False`.

L'ordre des résultats

La classe `Meta` est également l'endroit idéal pour définir l'ordre dans lequel vous souhaiterez voir les résultats s'afficher. L'option la plus simple pour cela est `ordering`, qui prend en argument une liste de la forme suivante :

```
ordering = ['pub_date','author']
```

Dans cet exemple, les résultats seront triés par date de publication (`pub_date`) et par auteur. Vous pouvez utiliser le signe - pour inverser l'ordre de tri.

C'est également via les options `Meta` que vous allez définir comment Django doit s'y prendre pour récupérer l'objet `latest`. Ce dernier est obtenu via la queryset suivante, qui doit vous renvoyer un seul objet :

```
last = Article.objects.latest()
```

L'option `get_latest_by` définit le champ grâce auquel Django vous renverra le dernier objet. La seule limitation est qu'il soit de type `DateTimeField` ou `DateField`.

Les contraintes d'unicité

Définir les contraintes d'unicité se fait habituellement au niveau des champs du modèle avec l'option `unique=True`. Pourtant, dans certains cas, c'est un ensemble de deux champs que vous souhaitez rendre unique, par exemple un nom et un prénom : plusieurs personnes peuvent avoir le même prénom, plusieurs peuvent avoir le même nom, mais vous ne voulez pas que deux personnes partagent à la fois le même nom et le même prénom. Dans ce cas, c'est l'option `unique_together` qu'il faut employer, de la façon suivante :

```
unique_together = ('nom', 'prenom')
```

Les permissions

Django vous autorise à définir des permissions supplémentaires dans vos modèles, que vous utiliserez comme les permissions habituelles. La syntaxe est la suivante :

```
permissions = (('peut_faire_quelquechose', 'Peut faire quelquechose' ),)
```

La première partie du tuple correspond au code de la permission et la seconde à la valeur affichée à l'écran des permissions. Nous en reparlerons en détail au chapitre 15 (section « Les permissions » page 345). Sachez simplement que vous pouvez utiliser dans vos vues :

```
user.has_perm('application.peut_faire_quelquechose')
```

Les méthodes prédéfinies

Lorsque vous manipulez vos modèles, vous utilisez sans y penser la méthode `save` sans avoir besoin de la définir et, la plupart du temps, cela suffit. En revanche, pour

réaliser des actions un peu complexes, cette méthode proposée par Django ne suffit plus. Il vous faudra alors définir votre propre méthode.

Prenons un exemple concret. Imaginons que vous souhaitiez recevoir un e-mail dès qu'un auteur publie un nouvel article. Voici le modèle `Article` :

```
class Article(models.Model) :
    titre = models.CharField(max_length=250)
    publication = models.BooleanField()
    contenu = models.TextField()
    sauteur = models.ForeingKey(Auteur)
```

Quand l'auteur va sauvegarder son article, vous vérifierez si `publication` a la valeur `True` et, si tel est le cas, vous déclencherez l'envoi d'un e-mail.

Comme la fonction `save()` existe déjà, commencez par écrire une nouvelle fonction `save`, qui ne fait rien d'autre qu'appeler celle de votre modèle :

```
def save(sef, *args, **kwargs) :
    super(Article, self).save(*args, **kwargs)
```

Validez ensuite que votre méthode fonctionne toujours aussi bien et qu'elle propose les mêmes fonctionnalités que le `save` d'origine. Vous n'avez donc plus qu'à vérifier la valeur de vos champs et à envoyer l'e-mail si la valeur de `publication` correspond bien à celle que vous attendez :

```
def save(sef, *args, **kwargs) :
    super(Article, self).save(*args, **kwargs)
    if self.publication is True :
        send_admin_mail(self)
```

La fonction `send_admin_mail` reste à écrire, mais ce n'est pas le sujet qui nous préoccupe ici. Nous vous laissons le soin d'écrire cette fonction, en sachant qu'elle reçoit comme argument un objet `Article` qui vous facilitera l'écriture d'un template pour votre e-mail.

La méthode `delete` fonctionne de la même manière. Ainsi, pour recevoir un e-mail lorsqu'un article est supprimé, il suffit d'écrire :

```
def delete(sef, *args, **kwargs) :
    send_admin_mail(self)
    super(Article, self).delete(*args, **kwargs)
```

La seule différence avec la méthode `save` que vous avez définie plus haut est que l'action est menée avant le `super`. Ici, la raison est évidente : l'objet n'existera plus une fois qu'il aura été supprimé.

Méthode

Comme vous avez pu le voir, les méthodes des modèles ne sont pas à négliger. Elle vous permettent de prendre le contrôle de votre base de données de bien des façons et de rendre votre application plus DRY et plus lisible. Dès que vous vous apercevez que le code de vos vues est un peu trop volumineux, posez-vous toujours la question suivante : « N'y a-t-il pas de la logique de base de données que je devrais déplacer dans mes modèles ? » Il ne fait nul doute que votre application sera alors beaucoup plus simple et plaisante à maintenir.

Les querysets

Vous avez déjà manipulé des enregistrements de base de données en utilisant des querysets, qui seront responsables en grande partie des performances de votre application. Il est donc important que vous connaissiez en profondeur leur fonctionnement pour en tirer le meilleur parti.

Description d'une queryset

Les querysets sont le cœur de l'ORM de Django. C'est à travers elles que vous manipulez les collections d'objets de votre base de données. Lorsque vous utilisez une queryset, vous faites en réalité appel à la base de données. C'est pourquoi, il est très important de bien en connaître le fonctionnement, afin d'optimiser les opérations en base de données.

Une queryset dans sa forme la plus simple se présente comme suit :

```
articles = Article.objects.all()
```

Cette écriture va potentiellement exécuter sur la base de données une requête de la forme suivante :

```
select titre,date, auteur, contenu from blog_article
```

Nous disons « potentiellement », parce qu'en réalité, Django va simplement enregistrer votre volonté de lancer cette commande. Cependant, si vous n'utilisez pas la variable `articles`, la requête ne sera pas exécutée. C'est le premier très gros avantage de la queryset : elle n'est évaluée, c'est-à-dire qu'elle fait un appel à la base de données, que lorsque cela est nécessaire. Votre travail d'optimisation va donc consister à exécuter votre queryset le plus tard possible.

Par exemple, si vous écrivez :

```
reponse = []
auteur = 'John'
articles = Articles.objects.all()
for article in articles :
    if article.auteur.name == auteur :
        reponse.append(article)
```

vous venez d'écrire de la pire façon qui soit… En effet, à la ligne `for article in articles :`, vous forcez Django à exécuter votre queryset. Vous allez donc rechercher tous les articles en base de données, ce qui risque d'être très coûteux. Et à la ligne `if article.auteur.name == auteur :`, vous allez chercher en base de données le champ `name` de l'auteur de l'article, une fois par article. Vous faites donc une requête sur l'ensemble des articles et une supplémentaire par article ; inutile de vous dire que vous allez très rapidement avoir des problèmes de performances.

En revanche, si vous écrivez :

```
articles = Article.objects.filter(auteur_name = "John")
```

vous ne faites qu'une seule requête en tout et pour tout, pour le même résultat. Toutefois, comme les querysets ne sont évaluées qu'au tout dernier moment, vous écrirez aussi bien :

```
articles = Articles.objects.all()
reponse = articles.filter(auteur_name= "John")
```

Les relations dans les querysets

Vos modèles sont reliés entre eux par des relations SQL et, très rapidement, vous voudrez profiter de ces relations pour filtrer votre contenu. Dans le cas d'une relation vers un objet, c'est assez simple et c'est exactement ce que vous venez de faire en écrivant :

```
articles = Article.objects.filter(auteur__name= "John")
```

En effet, vous avez sélectionné les objets de type `Article` dont l'auteur (dans une autre table) possède un champ `name` valant « John ». Notez au passage l'utilisation du double caractère de soulignement pour traverser une relation.

Que se serait-il passé si vous aviez souhaité faire la même opération dans l'autre sens, c'est-à-dire sélectionner tous les articles de John ? Cette fois, il faudrait écrire :

```
articles = Auteur.objects.get(name= "John").article_set.all()
```

Dans cette écriture, `article_set.all()` est une queryset, ce qui signifie que vous pouvez facilement filtrer encore plus avant cette requête, en écrivant par exemple :

```
latest = Auteur.objects.get(name=John).article_set.latest()
```

Cela aura pour effet de vous retourner le dernier article de John, en une seule requête.

En réutilisant ce que vous avez appris à propos des méthode des modèles, vous pouvez écrire une nouvelle méthode de `Auteur` :

```
def latest_article(self) :
    return self.article_set.latest()
```

et écrire dans votre vue :

```
latest = Auteur.objects.get(name=John).latest_article()
```

Le coût des querysets

Vous l'aurez compris, toutes les querysets ont un coût. Mais il existe quelques astuces pour le limiter.

Tout d'abord, n'exécutez vos querysets que lorsque cela est absolument nécessaire. Par exemple, renvoyer une queryset au template autorise à ne l'exécuter que dans certains cas. Prenons l'exemple suivant :

- dans votre vue :

```
context['articles'] = Article.objects.all()
```

- dans votre template :

```
{% if user.username == « John » %}
{% for article in articles %}
{{article.titre }}
{% endfor %}
```

Cette écriture n'exécute la requête que si l'utilisateur connecté est bien « John ». Vous pouvez aller plus loin. Si la seule chose dont vous ayez besoin dans votre template est le titre de l'article, ne demandez que le titre de l'article ! Par exemple, dans votre vue :

```
context['articles'] = Article.objects.values('titre')
```

Profitez également du système de cache des querysets. Par exemple, si vous écrivez :

```
titres = [article.titre for article in article.objects.all()]
contenus = [article.texte for article in article.objects.all()]
```

vous allez faire deux requêtes dans votre base de données. En revanche, vous ne faites plus qu'une seule requête si vous écrivez :

```
articles = articles.objects.all()
titres = [article.titre for article in articles]
contenus = [article.texte for article in articles]
```

Cette astuce peut également être utilisée dans vos templates avec l'instruction `with` :

```
{% with articles as arts %}
{% for article in arts %}
{{article.titre}}
{% endfor %}
{% for article in arts %}
{{article.text}}
{% endfor %}
{% endwith %}
```

Les managers

Ne vous êtes-vous jamais demandé pourquoi vous deviez écrire `Article.objects.all()` et non pas, tout simplement, `Article.all()` ? En fait, `objects` est un manager, c'est-à-dire une interface qui va se connecter pour vous à la base de données pour aller chercher le contenu que vous recherchez.

`Article.objects` est un manager, `Article.objects.all()` est une queryset. Le manager manipule pour vous des querysets.

Le fonctionnement des managers

Vous avez déjà rencontré deux types de managers fournis par Django : `objects` lorsque vous écrivez `Article.objects.all()` et `article_set` lorsque vous écrivez `Auteur.article_set.all()`. Ces managers possèdent une méthode `get_query_set()` qui accepte divers paramètres et qui renvoie une queryset.

Les autres méthodes de ces managers, vous les connaissez : `filter`, `get`, `exclude`... Elles ne font qu'une chose : prendre les paramètres que vous leur donnez et les trans-

mettre à `get_query_set()`, qui elle-même va retourner un objet de type queryset instancié avec les paramètres transmis. Par exemple :

```
Article.objects.all()
```

va invoquer la méthode `all()` du manager `objects`. Cette méthode va retourner le résultat de la fonction `get_query_set()` sans paramètres (vous n'avez rien filtré). Cette dernière va, elle, retourner un objet de type queryset avec en argument le nom du modèle que vous avez utilisé (ici, `Article`).

Les managers ne sont donc qu'une couche d'abstraction, un liant entre vos modèles et la classe `QuerySet` de Django.

Faire ses propres managers

L'intérêt principal de connaître le fonctionnement précis des managers est de pouvoir créer les vôtres afin d'ajouter de nouvelles méthodes au manager et de modifier le comportement des managers.

Ajouter de nouvelles méthodes

Le premier point important quand vous ajoutez de nouvelles méthodes à votre manager est de savoir que vous pouvez retourner ce que vous voulez ; vous n'êtes pas du tout obligé de retourner une queryset. Le manager de base (`objects`) utilise cette possibilité lorsque vous utilisez sa méthode `count` qui retourne un entier et la méthode `get` qui retourne un objet par exemple.

Pour enrichir un manager avec de nouvelles méthodes, il suffit de créer une classe qui hérite du manager de base et de l'utiliser dans vos modèles. Par exemple :

```
from django.db import models

class CustomManager(models.Manager) :
    def add(self, a, b) :
        return a + b
```

Ensuite, importez votre nouveau manager dans vos modèles et utilisez-le dans le modèle de votre choix :

```
from mymanagers import CustomManager

class MyModel(models.Model) :
    objects = CustomManager()
```

Employez enfin votre nouvelle méthode :

```
>>> MyModel.objects.add(1, 3) :
4
```

Évidemment, ceci n'est qu'un exemple pour comprendre le principe général de la création de nouvelles méthodes pour vos managers. Nous comptons sur vous pour créer des méthodes plus utiles.

Modifier le fonctionnement du manager

Ici, l'idée n'est pas d'ajouter de nouvelles méthodes à votre manager, mais de modifier celles existantes. Comme nous l'avons vu, la méthode `get_query_set` des managers est responsable de la création d'une queryset. Cette méthode va donc pouvoir être modifiée pour changer l'ensemble des méthodes de votre manager.

Par défaut, le manager de base d'un modèle contient l'ensemble des enregistrements présents en base de données. Par exemple, `Article.objects.all()` va renvoyer l'ensemble des articles enregistrés. En modifiant la méthode `get_query_set()` d'un modèle, vous aurez la possibilité de définir votre propre manager qui sera utilisé en lieu et place de celui fourni par défaut.

On commence donc de la même manière que pour ajouter une méthode supplémentaire à un manager :

```
from django.db import models
class CustomManager(models.Manager) :
```

Cette fois, c'est directement la méthode `get_query_set` qui va être utilisée, par exemple :

```
from django.db import models
class CustomManager(models.Manager) :
    def get_query_set(self) :
        return super(CustomManager, self).get_query_set().filter(published=True)
```

Avec cette méthode, vous avez maintenant la possibilité de modifier directement votre manager par défaut :

- soit en supprimant directement sa référence :

```
class Article(models.Model) :
    objects = CustomManager()
```

- soit en ajoutant un manager supplémentaire :

```
class Article(models.Model) :
    objects = models.Manager()
    published = CustomManager()
```

Dans le second cas, quand vous écrirez `>>> Article.objects.all()`, vous obtiendrez l'ensemble des objets de type `Article`, publiés ou non. Et quand vous écrirez `>>> Article.published.all()`, vous n'obtiendrez que les objets de type `Article` ayant été publiés.

Un exemple complet pour comprendre l'intérêt des managers

Afin de vous faire mieux comprendre l'intérêt des managers, voyons comment les utiliser avec des proxy, que nous avons déjà abordés, pour mettre en place un système de workflow.

Tout d'abord, vous aurez besoin d'un modèle. Pour cet exemple, il s'agit d'un objet de type `Location`, qui représente un bien immobilier à la location :

```
class Location(models.Model):
    """
    Un bien en location
    """
    published = QueryManager()
    objects = models.Manager()

    room_count = models.PositiveIntegerField(max_length=2)
    title = models.CharField(max_length=250)
    description = models.TextField()
    workflow_state = models.IntegerField(max_length=1,
                                         choices=(
            (0, 'brouillon'),(1,'soumis à publication'), (2, 'publié')))
```

Comme vous le voyez, nous utilisons le manager `QueryManager`, qu'il nous faut définir :

```
class QueryManager(models.Manager):
    def get_query_set(self):
        return super(QueryManager, self).get_query_set().filter(
            workflow_state=2)
```

Ce manager va retrouver uniquement les biens publiés (dont l'état de workflow est à 2). Dans l'ensemble du site, nous pourrions donc utiliser l'instruction suivante pour retrouver l'ensemble des locations publiées :

```
Location.published.all()
```

Le système de workflow veut que certains utilisateurs puissent éditer les locations qui sont à l'état de brouillons et les proposer pour validation à l'équipe de correcteurs. Pour cela, définissons un proxy du modèle`Location` :

```
class LocationProxy(Location):
    objects = EditorManager()
    class Meta:
        proxy = True
```

Cette fois, nous avons redéfini le manager `objects`. Le pseudo-modèle `LocationProxy` ne peut donc utiliser que ce manager, qu'il faut définir :

```
class EditorManager(models.Manager):
    def get_query_set(self):
        return super(EditorManager, self).get_query_set().filter(
            workflow_state__lte=1)
```

Lorsque l'on fera une queryset sur les objets de type `LocationProxy`, on ne récupérera que ceux dont l'état de workflow est à 0 ou 1, c'est-à-dire `brouillon` ou `soumis à publication`. Pour que le système soit vraiment utile, il faut encore faire en sorte que les « éditeurs » ne puissent pas choisir l'état `publié` sur l'un des objets. En effet, comme `LocationProxy` n'est rien de moins qu'une copie du modèle `Location`, son formulaire reste identique à celui de son modèle parent.

Créons le formulaire suivant :

```
class EditableLocationForm(forms.ModelForm):
    workflow_state = forms.ChoiceField(
        choices = (
        (0, 'brouillon'),(1,'soumis à publication')
                    )
        )

    class Meta:
        model = LocationProxy
```

Ce formulaire ne proposera donc que deux choix possibles : `brouillon` ou `soumis à publication`. Dans le fichier `admin.py`, il suffit de définir une classe `LocationProxyAdmin` qui utilise ce formulaire :

```
class LocationProxyAdmin(admin.ModelAdmin):
    form = EditableLocationForm
```

L'objet Q : encapsuler des paramètres pour exécuter des requêtes

Les filtres que vous employez dans la création de vos querysets sont le plus souvent suffisants pour vos besoins. Vous pouvez filtrer de nombreuses façons et obtenir des résultats satisfaisants, même si votre requête est complexe. Pourtant, les querysets souffrent d'un défaut : elles ne supportent que l'opérateur ET. Ainsi, quand vous écrivez :

```
>>> articles.objects.filter(published=True).filter(
   author__name='John').exclude(
    published_date__gte = datetime.datetime.now())
```

cela se traduit en langage courant par « articles parus ET dont le nom d'auteur est John ET dont la date n'est pas supérieure ou égale à maintenant ».

En revanche, si vous souhaitez écrire « articles parus ET dont l'auteur est John OU Pierre » le moteur de querysets s'effondre alors immédiatement. Heureusement pour le développeur Django, il n'est alors pas nécessaire de revenir à la méthode consistant à écrire du SQL. Django en offre une simple et puissante pour gérer ce type de cas : l'objet Q.

Description de l'objet Q

L'objet Q sert à encapsuler des paramètres pour exécuter des requêtes. Il peut être utilisé seul ou conjointement avec une queryset. Il donne accès aux opérateurs booléens tels que AND, OR, NOT, NOR...

Vous instanciez un objet Q de la façon suivante :

```
>>> pub = Q(published=True)
```

Vous pouvez ensuite utiliser cet objet dans une queryset classique, par exemple :

```
>>> articles = Article.objects.filter(pub)
```

ou

```
>>> articles = Article.objects.filter(Q(published=True))
```

ces deux écritures étant équivalentes à :

```
>>> articles = Articles.objects.filter(published =True)
```

Ceci étant, dans l'exemple que nous venons de voir, l'objet Q n'offre rien de plus qu'une queryset classique. C'est lorsque vous utilisez ensemble plusieurs objets Q que vous êtes en mesure d'en tirer vraiment parti.

Reprenons l'exemple « articles parus ET dont l'auteur est John OU Pierre ». Vous pouvez, grâce à l'objet Q, l'écrire comme ceci :

```
>>> Article.objects.filter(Q(auteur='John') |
Q(auteur='Pierre')).filter(published=True)
```

Dans cet exemple, vous avez créé deux objets Q :

```
Q(auteur='John')
```

et :

```
Q(auteur='Pierre')
```

Le signe | signifie « ou » et le signe & « et ». Enfin, vous pouvez utiliser le signe ~ pour inverser l'objet Q. De cette façon :

```
Q(auteur ='Pierre' ) | ~Q(auteur='John')
```

signifie « sélectionner les enregistrements où l'auteur est Pierre ou n'est pas John ».

Vous pouvez également écrire :

```
~Q(auteur ='Pierre' ) & ~Q(auteur='John')
```

qui signifie « sélectionner les enregistrements où l'auteur n'est ni Pierre ni John ».

Quand faut-il utiliser l'objet Q ?

Dès que vous pouvez utiliser une queryset classique, ne vous en privez pas. Généralement, sa syntaxe est plus claire et plus facile à mettre en œuvre. En revanche, si votre code est plus lisible avec des objets Q, n'hésitez pas à vous en servir. Il faut de toute façon savoir qu'en interne, Django utilise l'objet Q même lorsque vous lancez une queryset « ordinaire ». Enfin, bien entendu, il est des cas où vous n'avez pas le choix. Par exemple, quand les opérateurs booléens sont les seuls à vous permettre d'exprimer votre queryset, l'objet Q est incontournable.

Conclusion

Les outils que vous offre Django pour manipuler vos enregistrements de base de données sont nombreux et puissants. C'est très souvent au travers d'eux que vous commencerez vos applications et généralement au niveau de la couche « base de données » que vous optimiserez votre code. Prenez donc le temps de bien connaître vos outils et maîtrisez-les du mieux que vous le pouvez.

11

Le traitement de l'information : les vues

Les vues sont la seconde partie du concept MVC, même si, comme nous avons pu le voir, il s'agit plus d'une sorte de contrôleur que réellement de la vue. En réalité, la place que les vues vont prendre dépendra surtout de l'architecture de votre application. La question qu'il faut se poser est la suivante : « Où réside la logique de votre application ? » Est-ce dans les vues ou dans les modèles ? Bien souvent, ce sera un peu dans les deux. Dans une application très simple, qui fait fortement usage des vues-classes, la logique métier de votre application sera très certainement dans les URL et vos vues se réduiront à la portion congrue. Au contraire, dans une application sans base de données, la logique de votre application résidera certainement dans vos vues.

Tour d'horizon des vues

Fondamentalement, les vues sont responsables d'un travail simple : créer un contexte, le charger dans un template et retourner un objet, généralement de type `HttpResponse`. Derrière ce concept basique, vous allez rencontrer un grand nombre de notions concernant le fonctionnement du Web : en-têtes HTTP, méthodes, codes de statut, etc.

Le workflow standard

Que se passe-t-il lorsqu'un internaute entre dans la barre d'adresse de son navigateur une URL qui correspond à votre application ?

1. Le serveur web de votre machine reçoit une requête. Il s'agit d'un ensemble contenant :
   - une méthode, qui peut être l'un des cinq verbes suivants : `GET`, `POST`, `PUT`, `DELETE` et `PATCH` (nous y reviendrons en détail dans la section « Les verbes du Web », page 315) ;
   - une adresse ;
   - un en-tête HTTP ;
   - en option, des données (un formulaire rempli par exemple).
2. Le serveur web va alors consulter une table de relations afin de savoir ce qu'il doit faire de la requête : soit la traiter lui-même (par exemple, dans le cas d'un fichier statique), soit la rediriger vers une application (votre application Django).
3. Django va à son tour consulter le fichier `urls.py` racine de votre application – ce fichier étant signalé dans le `settings.py` de votre application –, puis le parcourir à la recherche d'une URL qui corresponde à la requête. Dès qu'il trouve une correspondance, la requête de l'utilisateur est transmise à la vue adéquate qui la traitera, puis renverra une réponse.

Durant ce processus, il y a plusieurs cas où l'utilisateur peut recevoir une erreur.

- Une erreur est retournée par votre serveur web : soit il ne trouve pas le fichier statique demandé, soit c'est l'applicatif susceptible de traiter la requête. Dans ce cas, le serveur web renvoie une erreur `404`, qui correspond au message `Page non trouvée` que vous avez peut-être déjà rencontré sur Internet.
- Il se peut que l'URL fournie par votre internaute ne corresponde à aucune URL de vos fichiers `urls.py` ; c'est alors Django qui se chargera de retourner une erreur `404`.
- L'URL a bien été trouvée, mais votre application retourne une erreur. Dans ce cas, il existe une petite subtilité. Soit le processus Django a levé une exception et c'est à lui qu'il incombe de renvoyer une erreur `500 Une erreur est survenue`, soit le processus Django est lui-même en erreur et n'est plus en mesure de répondre et ce sera donc à votre serveur web de fournir l'erreur `500` à l'internaute.

L'objet request

Bien entendu, dans la très grande majorité des cas, la requête de l'internaute est arrivée jusqu'à l'une de vos vues. C'est à vous maintenant de l'analyser, de la traiter et de transmettre à l'internaute une réponse qui lui convienne. Pour vous simplifier la vie, Django

ne vous transmet pas la requête brute de l'internaute, mais un objet de type `HttpRequest` contenant toutes les informations dont vous avez besoin. En passant dans les `urls.py`, Django aura également traduit certaines parties de l'URL pour vous fournir des paramètres que vous utiliserez directement dans votre vue. Par exemple, l'URL :

```
url(r'^/article/(?P<pk>\d+)/$, ma_vue)
```

appellera la fonction `ma vue` avec le paramètre `pk`.

Un appel à `http://monserveur.com/article/1/` est l'équivalent de :

```
>>> ma_vue(request,pk=1)
```

Bien souvent, les seuls paramètres fournis par l'internaute vous suffiront pour lui renvoyer le résultat qu'il attend. Pourtant, l'objet `HttpRequest` qui vous est fourni dans chacune de vos vues (`request`) rend des services très intéressants et vous aide à sélectionner très finement les données que vous allez renvoyer à l'utilisateur.

La méthode

Chaque requête doit posséder une méthode. Très naturellement, l'objet `request` qui est fourni à votre vue vous indique quelle méthode est utilisée par l'internaute. L'un des cas d'utilisation les plus courants de la méthode est dans la gestion des formulaires. Si la requête est effectuée avec la méthode `GET`, vous afficherez le formulaire ; si elle l'est avec la méthode `POST`, vous le validerez :

```
def gestion_form(request) :
    if request.method == 'POST' :
        # Ici vous allez valider les données
        # présentes dans le formulaire
    else :
        # La requête n'a pas été
        # effectuée avec la méthode POST,
        # vous allez donc afficher un formulaire vide
```

Accéder à la méthode d'une requête est assez simple. Cependant, vous avez également vu que la requête pouvait contenir des données. Dans le cas de la validation d'un formulaire, il va être extrêmement important de pouvoir accéder à ces données afin de les manipuler. Pour cela, Django met à votre disposition un dictionnaire `POST` pour les requêtes émises avec la méthode `POST` et un dictionnaire `GET` pour les requêtes émises avec la méthode `GET`. Vous pouvez alors facilement chercher dans ce dictionnaire la ou les données dont vous avez besoin, par exemple :

```
name = request.POST['name']
```

En revanche, n'allez pas trop vite et vérifiez bien la méthode de votre requête avant de chercher une donnée dans le dictionnaire de la méthode. En effet, le dictionnaire `POST` n'existe pas si la requête a été émise avec un autre verbe et Django lèvera alors une exception.

Les en-têtes

Parfois, accéder à la méthode ne vous suffira pas et vous aurez besoin également des en-têtes de la requête pour savoir comment traiter la demande. Si vous souhaitez traiter la requête en fonction du navigateur web de l'internaute qui visite votre site, par exemple en proposant une version du site pour les mobiles et une autre pour les ordinateurs, vous aurez besoin d'accéder à l'information `user agent` de la requête. Cela fait partie des informations de base fournies directement par le serveur web employé. Cet ensemble d'informations de base se trouve dans l'attribut `META` de l'objet `HttpRequest`. Cet attribut est un dictionnaire ; il vous suffit donc de demander la clé dont vous avez besoin, `HTTP_USER_AGENT` dans notre exemple (sur un Mac, depuis le navigateur Firefox).

```
print request.META['HTTP_USER_AGENT']
>>> 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:14.0) Gecko/20100101
Firefox/14.0.1'
```

Bien d'autres données sont présentes dans les en-têtes et n'attendent que vous pour être utilisées : les données de cache, les cookies, etc. La manière de s'en servir est toujours identique à celles que nous venons d'expliquer.

Les informations fournies par Django dans la requête

Pour vous faciliter la vie, Django vous propose de nombreuses autres informations dans la requête. Elles ne proviennent pas directement de la requête émise par l'internaute lui-même, mais elles vous sont proposées par Django au travers de divers middlewares (voir plus loin pour plus d'informations à ce sujet). Sachez déjà qu'il s'agit de morceaux de code s'exécutant avant que la requête ne parvienne à la vue et en modifiant le contenu.

Lorsque vous utilisez la configuration de base de Django, plusieurs middlewares sont déjà actifs et vous permettent de profiter d'informations supplémentaires présentes directement dans la requête. Par exemple, si vous utilisez l'application `auth` des contribs, vous avez à votre disposition l'objet `user` de l'utilisateur actuellement connecté, directement dans la requête. Pour en disposer, l'instruction suivante est suffisante :

```
request.user
```

L'objet response

Si vous recevez toujours un objet `request`, il est de votre devoir de transmettre une réponse. La plus simple que vous puissiez fournir est certainement un objet du type `HttpResponse`, qui renvoie un code d'erreur `200` (ce qui correspond à un succès et un contenu optionnel). Sans plus de précisions, il s'agira d'un document de type texte `text/html`. Par exemple :

```
from django.http import HttpResponse
a = HttpResponse()
return a
```

Ce code aura pour effet de renvoyer à l'utilisateur une page vide (vous n'avez pas précisé de contenu) sans erreur. Vous pouvez bien évidemment renvoyer du contenu :

```
a = HttpResponse('<h1>Hello World</h1>')
```

ou un tout autre type de fichier (par exemple du JSON) :

```
import json
resp = {'a':'une valeur','b': 'une autre valeur'}
return HttpResponse(json.dumps(resp), content_type="application/json")
```

Les codes HTTP fournis par Django

L'objet `HttpResponse` est donc très simple. Pourtant, il ne peut fournir qu'une réponse de type `200`. Si vous souhaitez renvoyer tout autre code d'erreur, vous devez faire appel à d'autres objets qui, héritant de la classe `HttpResponse`, lèveront d'autres codes d'erreur. Django vous en propose plusieurs que vous pourrez facilement utiliser. Par exemple :

- `HttpResponseNotFound` : si vous connaissez déjà les codes d'erreur du protocole HTTP, vous savez qu'il correspond à une page non trouvée. C'est un code d'erreur normal en ce sens que l'utilisateur tente d'accéder à une ressource qui n'existe pas ;
- `HttpResponseServerError` : c'est votre application qui a retourné une erreur. Un problème est survenu de votre côté et en tout état de cause, vous devriez le corriger ;
- `HttpResponseForbidden` : l'utilisateur tente d'accéder à une ressource qui lui est interdite.

Les autres codes HTTP

Il existe de nombreux autres codes HTTP dont vous pourrez avoir besoin. Par exemple, dans le cas où vous fournissez une API REST, lorsqu'un appel est lancé pour créer une nouvelle ressource, vous devez renvoyer un code HTTP `201`. Pour cela, c'est très simple : vous n'êtes même pas obligé de créer une nouvelle classe. En effet, la classe `HttpResponse` peut prendre en paramètre optionnel un `status_code`. Ainsi, vous pouvez écrire :

```
response = HttpResponse(status_code = 201)
```

Nous vous invitons à consulter la liste complète des codes HTTP sur le site http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

La compilation du template

Dans sa forme la plus simple, l'objet `HttpResponse` renvoie à l'utilisateur une chaîne de caractères, un code HTTP et un type MIME. Ce dernier est utile au navigateur pour savoir comment traiter la chaîne de caractères. La plupart du temps, il s'agira du type `text/html`. Vous le trouverez également sous le vocable `content-type`.

Il est évident que pour un framework tel que Django, renvoyer une chaîne de caractères ne suffit pas. Il vous faut bien plus pour mettre en place une architecture simple à maintenir et DRY. Ce dont vous avez besoin, c'est d'un moteur de templates. Il en existe de nombreux pour Python et vous pouvez tout à fait décider d'utiliser un autre moteur que celui que vous fournit Django. Cependant, dans les exemples qui vont suivre, c'est ce dernier que nous emploierons car il est particulièrement bien pensé pour des applications de taille conséquente.

Le principe de base de la compilation d'un template

Pour vous expliquer le principe de base de la compilation d'un template, nous allons prendre l'exemple d'une réponse HTML, car c'est le cas le plus courant. Pourtant, vous pouvez utiliser ce moteur de templates et le principe de la compilation pour absolument tout : XML, JSON, CSV, etc.

Le concept est donc de prendre un fichier statique, ici HTML, qui va contenir des « trous », un peu comme un modèle de lettre papier. Par exemple :

```
Le <date>
à <destinataire>
                <objet>
    <message>
```

Comme vous le voyez, pour que cette lettre soit lisible, il faudrait remplir les champs `date`, `destinataire`, `objet` et `message` avec de vraies données. Vous pourriez, par exemple, créer un dictionnaire Python qui remplirait ce rôle :

```
{'date' : '13 Juin 2012', 'destinataire' : 'Jean Dupond', 'objet' : 'Mon premier
projet', 'message' : 'Bonjour, je me permets de vous contacter aujourd'hui
car...'}
```

Vous n'auriez plus qu'à remplir votre lettre avec le contenu de votre dictionnaire. Eh bien, c'est exactement ce que vous propose Django !

Votre lettre, en langage Django, est un template. Il s'agit d'une classe que vous fournit le framework à qui vous allez simplement donner votre fichier, par exemple :

```
template = Template('ma_lettre.html')
```

Il vous faut ensuite fournir les données que vous allez insérer dans ce template. Il s'agit de la classe `Context` que vous manipulerez comme un dictionnaire :

```
contexte = Context({'date' : '13 Juin 2012', 'destinataire' : 'Jean Dupond',
'objet' : 'Mon premier projet', 'message' : 'Bonjour, je me permets de vous
contacter aujourd'hui car...'})
```

Il vous suffit maintenant de charger les informations du contexte dans le template :

```
template.render(context)
```

La méthode `render` de votre objet `template` se chargera de vous renvoyer votre fichier `ma_lettre.html` rempli avec votre contexte. C'est aussi simple que cela. Cette chaîne de caractères peut ensuite être retournée à l'utilisateur dans un objet de type `HttpResponse`. Bien entendu, il peut être fastidieux de créer pour chacune de vos vues un contexte et un template, puis d'utiliser `template.render(context)` et enfin de renvoyer à l'utilisateur un objet de type `HttpResponse` contenant le template rempli. C'est pour cela que Django vous propose une écriture simplifiée avec `render_to_response`.

render_to_response

`render_to_response`, dans `django.shortcuts`, est une écriture simplifiée de ce que nous venons de voir. Son fonctionnement consiste à renvoyer un objet de type `HttpResponse` contenant un template chargé avec un contexte.

Pour reprendre l'exemple précédent, en utilisant la méthode « à l'ancienne », vous écrirez :

```
template = Template('mon_template.html')
contexte = Context(un_dictionnaire)
response = template.render(context)
return HttpResponse(response)
```

En utilisant `render_to_response`, vous obtiendrez le même résultat en écrivant :

```
return render_to_response('mon_template.html',un_dictionnaire)
```

Il reste un petit problème néanmoins. Si vous utilisez l'application `auth` des contribs ou d'autres fonctionnant sur le même principe, il y a des éléments présents dans l'objet `request` dont vous aurez besoin dans votre template, par exemple la variable `user` pour l'application `auth`, ou des aides précieuses comme `STATIC_URL`. Dans les exemples que nous venons de voir, ces données ne sont pas renvoyées au template, qui ne peut donc pas les utiliser. Il faut par conséquent renvoyer également l'objet `request` dans l'objet `HttpResponse`.

L'objet `Context` n'est pas en mesure de renvoyer l'objet `HttpRequest` qui a été fourni à votre vue. Toutefois, Django vous propose un nouvel objet, `RequestContext`, qui fonctionne de la même manière que `Context` et qui accepte de prendre en argument l'objet `request` et de le renvoyer au `Context`.

L'objet RequestContext

L'objet `RequestContext`, dans `django.template`, peut donc recevoir l'argument `request` et le fournir au template. Son fonctionnement est identique à celui de l'objet `Context`, à la différence qu'il prend donc en argument un objet de type `HttpResponse`. Son utilisation est la suivante :

- en écriture « à l'ancienne » :

```
template = Template('mon_template.html')
contexte = RequestContext(request,un_dictionnaire)
response = template.render(context)
return HttpResponse(response)
```

- en écriture raccourcie :

```
return render_to_response('mon_template.html',un_dictionnaire,
context_instance=RequestContext(request)
)
```

Dans la première écriture, nous avons simplement utilisé `RequestContext` au lieu de `Context`, en lui fournissant comme premier argument l'objet `request`. Dans le second cas, nous avons ajouté un paramètre `context_instance` dont la valeur est un objet `RequestContext` contenant l'objet `request`. Cette écriture est malheureusement beaucoup moins simple que les précédentes. Sachant que l'action de renvoyer la requête au template va être effectuée très souvent, Django propose une écriture plus simple pour renvoyer un objet `HttpResponse` chargé avec un template, un `Context` et l'objet `request` : il s'agit de `render`.

render

`render`, dans `django.shortcuts`, va donc renvoyer tout ce dont l'utilisateur a besoin : l'objet `request`, un template et les données à charger dans le template. Son écriture est très simple :

```
return render(request, 'mon_template.html', mon_dictionnaire)
```

Ces raccourcis, `render-to-response` et `render`, sont extrêmement pratiques. Vous verrez que dans le développement de votre application, vous utiliserez principalement `render`. Pourtant, gardez à l'esprit le fonctionnement interne de la compilation du template, car vous en aurez malgré tout besoin, soit si vous ne souhaitez pas renvoyer l'objet `request` (cela a malgré tout un coût en termes d'appel en base de données), soit si vous renvoyez un code HTTP spécifique ou si souhaitez avoir un plus grand contrôle sur l'objet `HttpResponse`.

Les context processors

Comme vous l'avez constaté, certaines variables ne sont disponibles que lorsque vous renvoyez l'objet `request` à la vue. Ces variables vous sont fournies par différentes applications, comme `django.contrib.auth`. Le mécanisme mis en œuvre par ces applications est celui des context processors. Ce sont des callables qui prennent en argument l'objet `request` et qui renvoient un dictionnaire qui sera ajouté au contexte. Dans le cas de l'application `django.contrib.auth`, il s'agit de la fonction `django.contrib.auth.context_processors.auth`. Il vous est par ailleurs possible de définir vos propres context processors au sein de vos applications, en ajoutant une fonction qui prend l'objet `request` en argument.

Pour vous en démontrer la facilité de mise en place, nous allons écrire un context processor affichant l'adresse IP de l'utilisateur qui charge la page. Tout d'abord, dans l'une de vos applications, créez le fichier `context_processors.py` et écrivez-y la fonction suivante :

```
def add_ip(request):
    return {'ip': request.META['REMOTE_ADDR']}
```

Cette fonction prend l'objet `request` en argument et renvoie un dictionnaire qui contient l'adresse IP de l'utilisateur. Ceci fait, il faut maintenant l'ajouter à `settings.py`. C'est ici que réside le piège. En effet, dans `settings.py` vous ne verrez pas de variable relative aux context processors, mais elle existe pourtant bel et bien : elle est chargée automatiquement par Django. Si vous écrivez :

```
TEMPLATE_CONTEXT_PROCESSORS = (<vos context processors>)
```

vous allez écraser les context processors définis par Django, ce qui est problématique. Une autre solution consisterait donc à lister tous ceux utilisés par Django et à ajouter les vôtres en fin de liste. Encore une fois, c'est une solution qui souffre de défauts car, au fil des versions, Django peut ajouter, modifier ou supprimer des context processors. Vous prenez donc le risque de rencontrer des problèmes qu'il vous sera difficile de débusquer.

La solution que nous vous proposons consiste donc à récupérer les context processors définis par Django et à ajouter les vôtres de la façon suivante :

```
import django.conf.global_settings as DEFAULT_SETTINGS
```

Cette ligne va rendre disponible au sein de `settings.py` la variable `DEFAULT_SETTINGS` définie par Django et cela, quelle que soit la version employée. Maintenant que vous avez la liste à jour des context processors, vous pouvez y ajouter le vôtre :

```
TEMPLATE_CONTEXT_PROCESSORS = DEFAULT_SETTINGS.TEMPLATE_CONTEXT_PROCESSORS + (
    "votre_application.context_processor.add_ip",)
```

Enfin, pour valider que votre context processor fonctionne, vous pouvez vous rendre dans le template de votre choix et y écrire :

```
<h1>Votre addresse IP est {{ip}}</h1>
```

Sous réserve que votre template soit bien appelé par une vue qui vous fournit l'objet `request`, vous verrez l'adresse s'afficher.

Les vues et les objets de modèles

Bien souvent, quand vous développez les vues, votre travail consiste à récupérer en base de données via une queryset les objets que vous souhaitez transmettre à votre template. Malgré le côté magique et très simple de la création des vues, nous vous conseillons de garder quelques éléments à l'esprit.

Quand vous renvoyez une queryset dans votre contexte, celle-ci n'est évaluée qu'au niveau du template. Cela signifie que c'est à la compilation de ce dernier que vont se

faire les appels en base de données. Si, par exemple, le template n'affiche que les 10 premiers éléments de votre queryset, Django n'ira chercher en base de données que les 10 premiers éléments. C'est une fonctionnalité très utile qui permet de gérer très finement les requêtes qui vont être effectuées et de faire de vraies gains de performance. Notre conseil sera donc le suivant : renvoyez autant que possible une queryset plutôt qu'un dictionnaire. Sauf que…

Si renvoyer une queryset présente des avantages, il y a aussi des inconvénients qu'il faut prendre en compte. Vous pouvez, au travers du template, accéder aux relations des objets que vous fournissez. Ainsi, la liste des articles donne aussi accès à celle de leurs auteurs. Si la personne en charge de la rédaction du template ne sait pas ce qu'elle fait, elle peut transformer une requête très peu gourmande comme `Article.objects.get(pk=1)` (soit le premier article enregistré en base de données) en `{% for art in article.author.articles %}` (soit l'ensemble des articles de l'auteur de l'article en cours). Dans un tel cas, il peut être intéressant de limiter les possibilités d'accès à la base de données dans le template en forçant l'évaluation de la queryset et en ne renvoyant au template que les informations réellement utiles.

En bref, il ne tient qu'à vous de choisir la meilleure approche en fonction du contexte dans lequel vous vous trouvez, l'important ici étant d'avoir toujours en tête ce qui se passe derrière le code que vous écrivez afin de garder le contrôle de vos ressources.

Les vues avancées

Au début du chapitre, vous avez vu comment manipuler vos vues. Vous avez appris à utiliser certains raccourcis proposés par Django et à manipuler les nombreuses informations présentes dans la requête pour renvoyer différents types de contenus. Avec ces connaissances, vous êtes en mesure de créer des applications entièrement fonctionnelles et performantes. Nous allons maintenant voir comment faire plus que du HTML avec les vues, et comment organiser votre espace de travail pour rester le plus DRY possible et faire évoluer votre application facilement.

Organiser les vues d'un projet

Au fur et à mesure du développement de votre application, vos fichiers `views.py` vont prendre rapidement de l'embonpoint, ce qui peut être un véritable frein à l'évolutivité de votre projet. Qui en effet a envie de se plonger dans la lecture d'un fichier de plusieurs milliers de lignes ? La solution est simple : découpez et, si possible, découplez !

Tout d'abord, il faut diviser votre projet en unités d'applications, qui doivent être pensées le plus possible pour être réutilisables dans vos futurs projets. Toutefois,

même si vous le découplez au maximum, certaines applications risquent de devenir difficilement maintenables tout simplement parce que, par exemple, elles proposent de nombreuses fonctionnalités autour d'une ressource unique.

Le concept d'inclusion d'URL de Django va servir à découper votre application à un niveau plus bas et vous permettre de créer des ensembles logiques qui seront alors plus simples à maintenir.

Cas particulier : Ajax

Dans les exemples précédents, vous avez toujours renvoyé du HTML. Un cas très courant auquel vous allez également être confronté est celui d'Ajax. Lorsqu'une requête est effectuée via Ajax, la réponse attendue est souvent au format JSON. Nous allons donc décrire comment, avec Django, vous pouvez facilement renvoyer du JSON et comment faire en sorte qu'il soit correctement interprété par le navigateur client.

Tout d'abord, revoyons les fondamentaux. Tout ce qu'attend Django pour fonctionner convenablement est que la vue renvoie un objet de type `HttpResponse`. Dans le cas où vous souhaitez renvoyer du JSON, il vous suffit donc de modifier le content-type de l'objet `HttpResponse` :

```
return HttpResponse(content_type = 'application/json', un_objet_json)
```

Cette écriture est tout à fait convenable et fera bien ce que vous en attendez. La première question qui se pose cependant est : comment savoir si la requête demande du JSON ? Autrement dit, comment savoir si la requête est en Ajax ?

Reconnaître une requête Ajax

La première solution est de regarder dans les en-têtes de la requête en cherchant l'attribut `HTTP_ACCEPT`. Dans le cas d'une requête Ajax, cet attribut aura la valeur `application/json`. Vous pouvez donc écrire :

```
if request.META['HTTP_ACCEPT'] == 'application/json' :
    # le reste de votre code
```

Vous pouvez également utiliser l'une des méthodes de l'objet `HttpRequest` : `is_ajax()`. Elle cherche pour vous l'attribut `HTTP_X_REQUESTED_WITH`, qui doit contenir la valeur `XMLHttpRequest` dans le cas d'une requête effectuée via Ajax. Aujourd'hui, les frameworks Ajax modernes utilisent tous cette méthode. Pourtant, si vous appliquez votre propre solution pour effectuer une requête Ajax, il vous faudra ajouter cet entête si vous souhaitez que cette méthode fonctionne.

Le contexte JSON

Maintenant que vous savez comment renvoyer un objet `HttpResponse` adapté à JSON et que vous savez reconnaître une requête Ajax, il faut encore pouvoir renvoyer un objet JSON. Heureusement, Python vous offre une bibliothèque fort pratique pour cela : la bien nommée `json`. Elle effectue la conversion Python vers JSON. Voici un exemple d'utilisation :

```
import json
response = json.dumps({'a' : 10, b :'15'})
```

La méthode `dumps` de JSON va transformer votre dictionnaire en objet JSON. Tant que vous renvoyez des dictionnaires ou des listes, cette méthode sera efficace. Pourtant, vous allez rapidement vous rendre compte que cette bibliothèque ne peut pas tout faire. En effet, si ce que vous renvoyez à l'utilisateur est un objet de type queryset ou une instance de l'un de vos modèles (ce qui est assez fréquent), cela ne fonctionnera pas. En effet, la bibliothèque `json` est incapable de convertir une méthode en JSON, que vous trouvez par exemple dans les relations entre les objets. Il faut donc faire en sorte de donner à la méthode `dumps` des données qu'elle soit en mesure de convertir. Encore une fois, cela est naturellement prévu dans Django.

serializer

Django vous propose une fonction simple pour transformer une queryset en un format de sérialisation comme le JSON :

```
from django.core import serializer
response = HttpResponse()
serializer.serialize('json', MonModel.objects.all(),stream=response)
```

Avec ce court extrait de code, vous avez rempli l'objet `HttpResponse` avec l'ensemble des objets de type `MonModel`. En revanche, pour renvoyer cette réponse à l'utilisateur, vous devez encore modifier le content-type de l'objet `HttpResponse` qui est par défaut `text/html` :

```
response.content_type = 'application/json'
return response
```

Les relations et JSON

Nous sommes maintenant en mesure de renvoyer au navigateur un objet de type JSON qu'il saura traiter comme tel. La dernière chose à laquelle il faudra faire attention concerne les relations entre les modèles. Par défaut, Django va représenter les relations

avec leurs identifiants uniques, ce qui convient dans de nombreux cas. Parfois cependant, vous aurez besoin d'en savoir plus sur une relation et souhaiterez peut-être afficher autre chose que simplement l'identifiant unique de la ressource reliée.

La logique de sérialisation de Django représente un modèle au travers de sa clé primaire. Néanmoins, vous avez la possibilité de définir une méthode `natural_key` qui va définir une autre façon de présenter ce modèle. Prenons un exemple :

```
class Author(models.Model) :
    name = models.CharField(max_length= 100)

class Article(models.Model) :
    author = ForeignKey(Author)
    title = models.CharField(max_length=100)
```

Si nous sérialisons l'objet `Article`, le JSON qui sera renvoyé contiendra une clé `author` dont la valeur sera l'identifiant unique de l'auteur auquel l'article est rattaché. En revanche, si nous créons la méthode `natural_key` sur notre objet `Author`, nous aurons une clé `author` comme précédemment, mais qui correspondra à la valeur de retour de `natural_key`. Par exemple, le code suivant retournera le nom de l'auteur au lieu de son identifiant :

```
def natural_key(self) :
    return self.name
```

Bien entendu, vous pouvez également être plus bavard et retourner une représentation exacte de l'objet `author` avec tous ses champs si vous souhaitez l'inclure dans chacun de vos articles.

Aller plus loin dans la sérialisation

Même si ce que vous venez de voir correspond à de nombreux cas, il arrive toujours des situations où renvoyer l'ensemble des champs d'un modèle n'est pas ce que l'on désire. En effet, il est parfois souhaitable de ne renvoyer qu'un sous-ensemble de champs. Avec les sérialiseurs de Django, cela est tout à fait possible et fort simple en utilisant le paramètre optionnel `fields`. Comme son nom l'indique, ce paramètre ne sélectionne qu'un sous-ensemble de champs à renvoyer à l'utilisateur sous la forme d'un tuple, comme il est courant de le faire dans les différentes parties de Django (dans l'`admin` notamment). Par exemple :

```
serializer.serialize('json', MonModel.objects.all(),stream=response,
fields=('name', 'content'))
```

Dans le cas du JSON, un autre élément peut poser problème. Si vous renvoyez des données qui contiennent des caractères Unicode, vous risquez de ne pas les voir s'afficher correctement. Pour éviter cela, il faut utiliser le paramètre `ensure_ascii=False` garantissant que le contenu sera bien conservé en UTF-8.

Dans les exemples que nous venons de voir, nous n'avons traité que le cas du JSON, car c'est aujourd'hui celui le plus courant. Mais tout ce que nous avons expliqué reste vrai au sujet des autres formats de sérialisation que sont XML et YAML. Cependant, dans le cas de ce dernier, vous devrez avoir installé la bibliothèque `PyYaml` afin que ce sérialiseur soit disponible.

Cas particulier CSV : format d'échange de données tabulaires

Le CSV *(Comma Separated Values)* est un format très pratique d'échange de données tabulaires. Il renvoie un ensemble de données séparées par des virgules, d'où son nom. Le fonctionnement de Django est particulièrement bien adapté à ce type de format et vous pouvez même profiter du langage de template pour vous simplifier la vie. Les choses que vous devez garder à l'esprit sont les mêmes que dans le cas du JSON, à savoir principalement de renvoyer le content-type correct (en l'occurrence `text/csv`).

Contrairement au JSON, le CSV n'est pas un format que le navigateur est en mesure de gérer ; un tableur comme LibreOffice Calc est alors bien mieux adapté. Il faut donc non seulement renvoyer le bon content-type, mais également indiquer au navigateur de renvoyer ce résultat à un tableur ou, si ce logiciel n'existe pas sur l'ordinateur de l'utilisateur, de lui laisser la possibilité de l'enregistrer sur son disque dur. Pour ce faire, c'est la même méthode que pour l'enregistrement de fichiers binaires (voir plus loin). Il faut fournir un attribut de type `Content-Disposition`, qui va donner au navigateur des informations supplémentaires sur la manière de traiter la réponse. En l'occurrence, nous souhaitons qu'il la traite comme un fichier binaire. Il faut donc indiquer :

```
response = HttpResponse(mimetype='text/csv', )
response['Content-Disposition'] = 'attachment; filename=unfichier.csv'
```

Nous venons d'ajouter une « pièce jointe » à la réponse, qui sera traitée comme telle par le navigateur. Il reste à remplir cette pièce jointe avec du contenu. C'est tout naturellement que nous allons utiliser la bibliothèque `csv`. Elle traite le CSV comme un objet de type fichier. Or, d'un point de vue interne, l'objet `HttpResponse` est un objet de ce type. Les problématiques rencontrées vont donc se situer du point de vue de la bibliothèque `cvs` et non pas du côté Django.

Le principe de CSV est d'écrire une ligne par objet représenté, tous les attributs étant séparés par des caractères de séparation, habituellement des virgules. Vous pouvez donc écrire :

```
writer = csv.writer(response)
writer.writerow(['a', 'b', 'c'])
writer.writerow(['d', 'e', 'f'])
```

et ainsi de suite.

Dans le cas courant où vous souhaitez traiter un ensemble d'objets de type `models`, vous aurez sans doute à l'esprit d'écrire un code proche de ceci :

```
writer = csv.writer(response)
for elem in MyModel.objects.all() :
    writer.writerow([elem.name, elem.date, elem.content])
```

C'est assez simple et limpide. Cependant, utiliser le moteur de templates de Django peut vous faciliter la vie. Dans ce cas, il est inutile d'employer le module `csv`.

Renvoyer du CSV avec le moteur de templates Django

La seule différence avec une réponse classique en HTML est la gestion des en-têtes :

```
response = HttpResponse(mimetype='text/csv', )
response['Content-Disposition'] = 'attachment; filename=unfichier.csv'
```

Le reste du code que nous allons écrire est on ne peut plus classique :

```
context = {'data' :MonModel.objects.all()}
template = 'mon_template.txt'
template.render(context)
```

Le template qui correspondra à cette vue peut être le suivant :

```
{% for object in data %}
''{{ object.name}}'', ''{{object.date}}'', ''{{object.content}}''
{% endfor %}
```

Afin d'éviter les soucis d'apostrophes, utilisez le template tag `addslashes` qui va se charger d'ajouter à vos données les caractères d'échappement nécessaires.

Cas particulier PDF

Maintenant que nous avons vu les méthodes pour renvoyer un format sérialisé comme du JSON ou un format de type fichier comme CSV, le cas particulier du PDF ne devrait pas vous sembler étranger. Comme pour le rendu CSV, c'est plutôt la bibliothèque PDF qui risque de vous donner du fil à retordre. Sachant que le type MIME d'un fichier PDF est `application/pdf` et qu'il se comporte comme un objet de type fichier, il est transparent qu'il faut commencer par écrire :

```
response = HttpResponse(mimetype='application/pdf')
response['Content-Disposition'] = 'attachment; filename=unfichier.pdf'
```

Pour le « côté Django », c'est terminé. Si vous avez déjà travaillé sur la création de fichiers PDF, la suite est identique à ce que vous avez déjà fait. Pour tester les exemples suivants, vous devez avoir installé sur votre machine la bibliothèque Python qui sert à écrire du PDF, à savoir `ReportLab`. Sinon, vous l'installerez, comme les autres modules tiers, via la commande :

```
pip install reportlab
```

Un exemple simple

`ReportLab` nécessite un objet de type fichier pour fonctionner normalement. Or, comme on l'a déjà vu dans le cas du CSV, l'objet `HttpResponse` est de type fichier. Écrivons donc l'exemple le plus simple :

```
from reportlab.pdfgen import canvas
response = HttpResponse(mimetype='application/pdf')
response['Content-Disposition'] = 'attachment; filename=unfichier.pdf'
p = canvas.Canvas(response)
p.drawString(100, 100, "Hello world.")
p.showPage()
p.save()
return response
```

Cela nécessite quelques explications : tout d'abord, vous importez le module `canvas` de `ReportLab`. Les deux lignes suivantes, vous les connaissez bien ; elles vont renvoyer au navigateur les informations complémentaires dont il a besoin pour gérer le fichier PDF. La suite du code est donc spécifique à `ReportLab`. L'objet `Canvas` de la bibliothèque `canvas` peut être vu comme l'espace sur lequel vous allez dessiner. Pour qu'il se comporte correctement, il faut lui donner en paramètre un fichier de sortie dans lequel il va pouvoir s'enregistrer. Comme l'objet `HttpResponse` est de type fichier, il suffira d'écrire directement dedans. La ligne suivante, `drawString`, écrit un texte sur

votre espace de dessin. Il faut lui préciser les coordonnées x et y ainsi que le texte à afficher. L'appel suivant, `showPage`, écrit effectivement dans l'objet `response`. Tant que cette méthode n'est pas appelée, les modifications que vous avez faites à votre espace de dessin restent en mémoire. Il faut donc les écrire dans l'objet `response`. Enfin, vous appelez `save` qui va fermer proprement le *buffer* de `canvas`.

Avant et après la vue : les middlewares

Vous avez vu à maintes reprises comment l'objet `request` passe à travers des URL, puis une vue et enfin, comment la vue renvoie un objet `response` au client. Ce système est particulièrement souple, mais il souffre d'une carence qui ne saute pas immédiatement aux yeux. Que faire si vous souhaitez intercepter ou modifier l'objet `request` avant qu'il n'atteigne les URL ou les vues ? Que faire si vous souhaitez modifier l'objet `response` avant qu'il ne soit renvoyé au client ? C'est là que les *middlewares* entrent en jeu. Ce sont des plug-ins qui vont se brancher sur le processus de `request`/`response`. Il s'agit de classes qui peuvent contenir jusqu'à cinq fonctions, chacune d'elles étant appelée à un moment spécifique du processus de `request`/`response`.

- `process_request` est exécutée au moment où Django reçoit une requête et doit décider qu'elle vue doit être utilisée.
- `process_view` est appelée au moment où Django appelle la vue. Vous avez donc à votre disposition la vue et tous les arguments qu'elle a reçus.
- `process_exception` : cette fonction est appelée lorsque la vue a lancé une exception.
- `process_template_response` : cette fois, la vue a été exécutée mais la compilation du template n'a pas encore eu lieu. Vous pouvez donc modifier le `Context`, changer le template…
- `process_response` : Django a terminé son travail. La réponse va être envoyée au navigateur. Cela vous laisse une dernière chance de modifier ce rendu d'une façon ou d'une autre avant qu'il ne soit définitivement renvoyé au navigateur client.

Bien entendu, Django implémente lui-même plusieurs middlewares. Leur liste est disponible dans la variable `MIDDLEWARE_CLASSES` du `settings.py` de votre projet. Pour créer les vôtres, il suffit donc d'écrire une nouvelle classe et de l'ajouter à cette liste.

Pour vous expliquer comment fonctionnent les middlewares, nous allons en créer un très simple : `RenderTimeMiddleware`. Il va calculer le temps nécessaire à Django pour exécuter une vue. Afin de rester simple et didactique, le résultat de ce calcul sera simplement affiché sur votre console de développement.

```
import datetime
class RenderTimeMiddleware(object):
```

```
import datetime
class RenderTimeMiddleware(object):

    def process_view(self, request, view_func, view_args, view_kwargs):
        request.time = datetime.datetime.now()

    def process_response(self, request, response):
        now = datetime.datetime.now()
        print now - request.time
        return response
```

Ajoutez simplement ce middleware à votre liste et rechargez l'une des vues de votre projet. Vous verrez alors s'afficher sur votre terminal le temps qu'il a fallu à Django pour la calculer.

En production, ce type de middleware vous sera très utile, mais sans doute faudrait-il utiliser un système de logs pour le rendre pérenne.

Les vues génériques

Les vues génériques existent depuis fort longtemps dans Django, mais leur utilisation ne s'était jamais vraiment démocratisée. C'est avec l'arrivée de vues génériques basées sur les classes qu'elles sont devenues une pièce centrale de l'échafaudage Django. Il est en effet plus rapide et plus simple de les employer dans votre projet. Et alors qu'elles étaient difficilement modifiables lorsque l'on souhaitait faire des choses non prévues par le framework, elles sont désormais pensées pour être héritées, modifiées, triturées. Cela tient au fait que les vues-classes ont été pensées avec une architecture extrêmement modulaire. Chaque vue générique n'est en fait que le résultat d'un ensemble de classes qui se « surhéritent ». Il est de ce fait particulièrement aisé d'aller chercher dans le code de Django la classe dont vous aurez besoin en fonction du caractère haut ou bas niveau de ce que vous recherchez.

Mais, au fait, que sont les vues génériques ? Il s'agit d'un ensemble de vues qui couvrent les besoins les plus courants dans le développement web : lister un ensemble d'objets, présenter un objet unique… Ces besoins courants représentent une grande part du développement web. Nous allons donc les disséquer, apprendre à les utiliser, mais surtout à en faire ce que l'on veut.

Un exemple simple : TemplateView

`TemplateView` est la plus simple des vues génériques que vous allez employer ; elle est parfaite pour découvrir le fonctionnement de ces dernières. Ici, pas de requête en

base de données, on souhaite renvoyer simplement un template statique. Par exemple, la page de login de votre application est un très bon exemple d'utilisation de `direct_to_template`. Son usage est tellement simple qu'il n'est même pas utile d'écrire quoi que ce soit dans les fichiers `views.py`. Vous l'utiliserez directement dans votre fichier `views.py` de la façon suivante :

```
from django.views.generic import TemplateView
from django.conf.urls import patterns, url, include
urlpatterns = patterns('',
url(r'^login/$',
    TemplateView.as_view(template_name='login.html')
```

Sans creuser dans les détails de l'implémentation de `TemplateView`, vous pouvez constater que la syntaxe est assez claire : vous employez la méthode `as_view` de `TemplateView` sans même l'instancier. Vous donnez en paramètre à cette méthode le nom du template que vous souhaitez renvoyer et Django se charge du reste. Si vous avez des objets dans l'objet `request`, l'utilisateur connecté par exemple, la variable (`{{user}}`) sera disponible dans le template. Ce type de fonctionnement est partagé par toutes les vues génériques et plus largement par toutes les vues-classes de Django.

Vous utiliserez toujours, dans vos URL, la méthode `as_view` avec ou sans arguments. En effet, avec les vues-fonctions, vous avez l'habitude d'instancier un contexte, puis un template, et de charger ce contexte dans le template, puis de renvoyer un objet `HttpResponse` contenant le template compilé avec le `Context`. Les vues génériques ont séparé chacune de ces actions dans des méthodes qui sont toutes appelées, dans l'ordre, avec la méthode `as_view`. C'est donc cette méthode qui va générer un objet `HttpResponse` et le renvoyer à l'utilisateur.

Ici, dans le cadre de `TemplateView`, seul le template reste à définir. Le `Context` est vide et l'objet `request` est renvoyé à l'utilisateur. Toutefois, vous n'êtes pas obligé de définir le template dans les URL. Si vous préférez le faire dans vos vues, il vous suffit de déterminer une nouvelle classe héritant de `TemplateView` :

```
#views.py
from django.views.generic import TemplateView
class LoginView(TemplateView)
    template_name = 'login.html'

#urls.py
from django.conf.urls import patterns, url, include
from views import LoginView
urlpatterns = patterns('',
url(r'^login/$',
    LoginView.as_view()
```

`TemplateView` ne prend pas de contexte. Mais attention, ce n'est pas tout à fait vrai… Les paramètres présents dans l'URL sont chargés dans le contexte `params`. Vous pouvez donc écrire des `TemplateView` génériques dont le contenu peut changer en fonction de ces `params`. Comme cette vue charge un contexte, il faut bien qu'il y ait une méthode pour le faire ; il s'agit de `get_context_data` que vous avez déjà rencontrée plusieurs fois. Vous pouvez donc tout à fait modifier cette méthode et faire bien plus qu'un simple `TemplateView`. Toutefois, prenez garde : si vous commencez à charger de nombreux éléments dans le contexte, il est fort probable que vous n'utilisiez pas le bon outil. Dans ce cas, il serait sans doute préférable d'employer l'une ou l'autre des vues génériques qui suivent.

ListView et DetailView

`ListView` et `DetailView` sont des classes que vous utiliserez vraisemblablement très souvent. Vous les avez déjà rencontrées plusieurs fois et elles sont à la base de bien des exemples de ce livre. Elles vont renvoyer respectivement une collection d'objets et un objet unique. Sans configuration particulière, sans héritage, voici comment les employer :

```
url(r'^articles/$', ListView.as_view(model=Article)
url(r'^artic
```

Ici, la seule différence entre ces deux vues génériques est le paramètre `pk` qui permet à DetailView de retrouver l'enregistrement qu'il faut renvoyer.

Vous pouvez également utiliser le champ `slug` si vous l'avez défini sur votre modèle pour retrouver l'enregistrement. Cela permet surtout d'écrire des URL plus « jolies » :

```
/article/django-le-framework-des-perfectionnistes-presses/
```

plutôt que :

```
/article/283636
```

Le nom du template que Django va chercher pour charger votre ou vos objets est déterminé par le nom du modèle, c'est-à-dire, dans les exemples que vous venez de voir, `article_list.html` et `article_detail.html`. Bien évidemment, il est tout à fait possible d'indiquer à Django d'utiliser votre propre template, exactement comme cela a été fait avec `TemplateView` :

```
url(r'^articles/$', ListView.as_view(model=Article,
template_name='articles.html')
```

Les clés du contexte sont créées en fonction de la classe employée : `object_list` dans le cadre de `ListView` et `object` dans le cadre de `DetailView`. Là encore, vous pouvez modifier tout cela avec le paramètre `context_object_name` :

```
url(r'^articles/$', ListView.as_view(model=Article,
template_name='articles.html', context_object_name='articles')
```

Cela vous permettra de créer un template `articles.html` :

```
{% for article in articles %}
{{ article.titre }}
{% endfor %}
```

Vous avez la possibilité de définir ces différents paramètres, soit directement dans les URL, comme précédemment, soit en surclassant les vues génériques. Pour reprendre les exemples précédents, vous pourriez écrire :

```
class ArticlesView(ListView) :
    template_name = 'articles.html'
    model = Article
    context_object_name = 'articles'
```

Dans vos URL, vous n'avez plus qu'à appeler `ArticleView.as_view()`.

ListView et queryset

Lorsque vous spécifiez un modèle en argument de votre `ListView`, Django crée pour vous une queryset. Sans plus de précision, il s'agit de `MonModel.objects.all()`. Pour la modifier, par exemple pour une `ListView` qui ne renvoie qu'un sous-ensemble d'objets, précisez le paramètre `queryset`. Reprenant l'exemple précédent, vous pouvez écrire :

```
import datetime
class ArticlesView(ListView) :
    template_name = 'articles.html'
    queryset = Article.object.filter(
                        date__lte = datetime.datetime.now()
                        )
    context_object_name = 'articles'
```

Ici, nous avons omis de préciser le paramètre `model`, qui n'est utile à Django que pour créer la queryset. Si vous lui fournissez directement cette dernière, il n'en a plus besoin. Parfois, en revanche, vos besoins sont plus génériques. Par exemple, vous voulez créer une `ListView` qui calculera elle-même la queryset à renvoyer. Dans ce cas, il vous faudra un peu plus de contrôle sur la manière dont Django détermine cette

dernière. Cela se fait simplement via la méthode `get_queryset` qui, comme son nom le laisse supposer, va créer la queryset en fonction de différents paramètres.

Comme vous êtes dans l'une des méthodes de `ListView`, vous avez accès à tous les attributs de votre classe, c'est-à-dire, entre autres : `self.args`, `self.kwargs` et, bien entendu, `self.request`. Votre méthode `get_queryset` peut ressembler à ceci :

```
def get_queryset(self) :
    return Article.objects.filter(author = self.request.user)
```

Cette méthode ne renverra que les articles de l'auteur actuellement connecté. Pour éviter que votre classe cesse de fonctionner lorsqu'un utilisateur non connecté essaie d'atteindre cette page, il faut pouvoir en limiter l'usage aux seules personnes déjà connectées. C'est un très bon exemple pour parler des décorateurs sur les vues génériques.

Les décorateurs et les vues génériques

Nous avons déjà vu les décorateurs sur les vues-fonctions. D'un usage très simple, on les utilise de cette manière :

```
@login_required
def ma_vue(request) :
…
```

Ici, nous utilisons `login_required`, décorateur issu de `django.contrib.auth` qui limite l'accès à la vue aux seuls utilisateurs enregistrés.

Dans le contexte d'une vue-classe (générique ou non), il y a deux manières d'utiliser les décorateurs. La première consiste à décorer les URL :

```
url(r'^articles/$', login_required(
                    ListView.as_view(
                       model=Article,
                       template_name='articles.html'
                        )
                     )
```

Ici, c'est `ListView.as_view()` qui est décorée. Le problème est que si vous employez régulièrement cette vue dans vos URL, il va être pénible de devoir utiliser un décorateur pour chacune d'entre elles : duplication de code, risque d'oubli, etc., ce n'est pas DRY.

La seconde méthode consiste à décorer la vue elle-même au niveau de la méthode `dispatch`, à l'aide de `method_decorator` :

```
from django.utils.decorators import method_decorator
from django.contrib.auth.decorators import login_required
```

```
class ArticlesView(ListView) :
    template_name = 'articles.html'
    model = Article
    context_object_name = 'articles'

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs) :
        return super(
            ArticlesView,self).dispatch(
                        *args, **kwargs)
```

Cette méthode, si elle est plus complexe à écrire que la précédente, en plus d'être plus DRY, présente également l'avantage de pouvoir aisément utiliser plusieurs décorateurs sur votre vue.

Les mixins

Certaines des méthodes disponibles pour `ListView` le sont aussi pour `DetailView`, et certaines autres sont spécifiques à l'une ou l'autre. Il serait fastidieux et contre-productif de vous lister, pour chacune des vues génériques, les différentes méthodes disponibles surtout qu'il existe par ailleurs un moyen bien plus simple.

Lorsque l'équipe de Django a implémenté les vues génériques, les développeurs se sont très vite aperçus que l'on pouvait découper les fonctionnalités en « blocs » partageant les mêmes méthodes. Au lieu donc de dupliquer la méthode `get_template`, par exemple, ils ont créé un ensemble de classes qui implémentent un cadre logique. Les vues génériques que vous manipulez sont donc ni plus ni moins qu'une agrégation de ces différentes classes : les *mixins*.

Si, au lieu de lister les méthodes des vues génériques, vous apprenez les différents mixins et leurs méthodes propres, il vous suffit ensuite de connaître quel mixin est employé pour telle ou telle vue générique, et vous savez exactement les méthodes qui sont disponibles. De plus, vous pouvez réaliser vos propre vues génériques en utilisant directement ces mixins et créer vos propres mixins pour modifier le comportement de certaines vues génériques.

Python, en tant que langage objet performant, autorise le système d'héritage multiple : une classe hérite de plusieurs autres classes, ce qui en fait une agrégation de ses classes parentes. C'est cette fonctionnalité qui permet de définir des vues-classes souples et simples d'emploi. Ainsi, en reprenant ce concept d'héritage multiple, `ListView` est créée à partir de `MultipleObjectTemplateResponseMixin` et de `MultipleObjectMixin`.

MultipleObjectMixin

Les méthodes de ce mixin sont créées dans l'optique où un ensemble d'objets doit être manipulé.

Pour déterminer les objets à afficher, `MultipleObjectMixin` a besoin que vous lui donniez une queryset. Pour cela, trois possibilités : soit avec la variable `model`, soit avec la variable `queryset`, soit avec la méthode `get_queryset`.

CreateView et DeleteView

Pour implémenter une interface CRUD, en plus de `ListView` et `DetailView`, vous avez besoin de pouvoir créer et supprimer des objets. Si la suppression est assez simple, pour la création d'objets en revanche, vous devez mettre en place un système assez lourd, avec un formulaire, une vue présentant le formulaire et une vue qui va récupérer les données du formulaire pour enregistrer le nouvel objet en base de données. Comme ce cas est assez courant, Django fournit une vue générique pour le faire beaucoup plus rapidement : `CreateView`.

CreateView

Son but est d'afficher un formulaire, de le traiter, de le renvoyer avec les erreurs éventuelles et d'enregistrer les données en base. Tout cela est naturellement proposé avec une écriture simplifiée :

```
url(r'^articles/new/$', CreateView.as_view(model = Article)
```

Elle est très similaire à celle de `DetailView` ou `ListView`. Pour que cette vue fonctionne, Django fait plusieurs assertions.

- Un template `nom_du_modele_create.html` existe.
- Le modèle employé dispose de la méthode `get_absolute_url` pour renvoyer l'utilisateur vers cette URL après l'enregistrement du formulaire.
- Bien entendu, vous disposez de la variable `template_name` si vous souhaitez utiliser un autre nom.

Le template CreateView

Le template dont `CreateView` a besoin pour fonctionner peut, dans sa forme la plus simple, ressembler à ceci :

```
<form action='' method='POST'>
{% csrf_token %}
{{form.as_ul}}
<input type= 'submit' value='enregistrer' />
</form>
```

Ici, la seule chose à savoir est que le formulaire que Django a créé est représenté par la variable du `contextform`.

Les paramètres optionnels de CreateView

Si l'écriture de l'URL générant le formulaire est très simple pour un cas comme celui que nous venons de rencontrer, il est tout à fait possible de modifier les choix que Django fait. Vous pouvez modifier le nom du template comme pour `DetailView` et `ListView` avec `template_name`, et calculer le nom du template avec la méthode `get_template`. Cette méthode devra retourner un nom de template valide, vous laissant le soin de le calculer.

Les autres vues génériques

Les vues génériques `ListView` et `DetailView` seront sans doute celles que vous aurez le plus besoin, à la fois parce que ce type de vues correspond à la palette d'usages la plus large, mais aussi parce qu'elles sont à l'origine des autres vues génériques.

Les vues génériques basées sur les dates

Un cas un peu particulier concerne les vues génériques basées sur les dates. Il s'agit d'un écosystème servant à naviguer dans une collection d'objets à partir d'un champ `date` ; ce peut être une date de parution par exemple. Nous avons déjà manié ces vues génériques dans l'application d'agenda partagé. Revenons un peu sur leur fonctionnement.

Tout d'abord, les vues génériques basées sur les dates sont des vues de collections comme les autres. Leurs possibilités sont les mêmes que celles offertes par `ListView` plus quelques autres. Pour comprendre leur fonctionnement, il faut en apprendre plus sur les mixins employés. Pour toutes les classes basées sur les dates, vous retrouvez `MultipleObjectMixin` en conjonction avec `DateMixin`.

DateMixin

`DateMixin` sert à manipuler des dates. Il offre la possibilité de configurer deux variables : `date_field`, qui correspond au champ date, et `allow_future` qui autorise ou non l'usage de date dans le futur. Elles sont adossées à deux méthodes : `get_date_field()` qui va calculer le `date_field` ainsi que `get_allow_future` qui va calculer l'autorisation d'employer des dates dans le futur.

`DateMixin` et `MultipleObjectMixin` sont utilisés dans une classe dont vont hériter toutes les classes basées sur les dates : `BaseDateListView`.

BaseDateListView

`BaseDateListView` implémente également quelques méthodes qu'il faut connaître pour faire un bon usage des vues génériques basées sur les dates :

- `get_dated_items()`, qui est une méthode renvoyant un tuple de trois éléments : une liste de dates, une liste d'objets et un `extra_context`. Ce contexte supplémentaire sera ajouté au `Context` fourni par `MultipleObjectMixin` ;
- `get_dated_queryset()`, qui va renvoyer la queryset disponible dans `get_dated_items()` ;
- `get_date_list()`, qui va retourner une liste de dates pour lesquelles un enregistrement existe.

Les vues génériques de dates de Django

Après ce rapide tour d'horizon du fonctionnement interne des vues génériques basées sur les dates, vous allez apprendre à vous servir des classes que Django a déjà écrites pour vous et qui sont directement utilisables.

ArchiveIndexView

Cette fonction affiche les derniers objets d'un modèle. Par défaut, elle ne présente pas ceux dont la date de référence est dans le futur. Cependant, avec `DateMixin`, vous pouvez spécifier `allow_future = True` pour modifier ce fonctionnement.

Dans sa forme la plus simple, `ArchiveIndexView` peut être implémentée comme suit :

```
ArchiveIndexView.as_view(
    model=MonModel,
    date_field='created_date')
```

L'argument `paginate_by`, qui est sur toutes les vues génériques de type liste (en réalité, toutes les classes qui héritent de `BaseListView`), vous sera particulièrement utile pour ne proposer, par exemple, que les cinq derniers articles d'un blog :

```
ArchiveIndexView.as_view(
    model=MonModel,
    date_field='created_date'
    paginated_by=5)
```

YearArchiveView

Comme son nom le laisse supposer, `YearArchiveView` présente des objets qui partagent la même date. Ici, l'implémentation va être la même que pour `ArchiveIndexView` :

```
YearArchiveView.as_view(
    model=MonModel,
    date_field='created_date'
    paginated_by=5)
```

Les autres vues-listes basées sur les dates

Il existe d'autres vues-listes basées sur les dates : `MonthArchiveView`, `WeekArchiveView`, `DayArchiveView` et `TodayArchiveView`. Leur fonctionnement est identique à celui de `YearArchiveView` et elles acceptent les mêmes options. La seule différence est le laps de temps concerné : mois, semaine, jour.

Les vues-classes

Dans les exemples précédents, nous avons toujours présenté des vues-fonctions, qui sont plus simples à comprendre et dont l'utilisation en ligne de commande reste là aussi facile. De plus, les vues-fonctions sont toujours d'actualité. En effet, il n'est pas prévu par l'équipe Django de les supprimer, ce qui est une très bonne chose car elles sont plus simples à comprendre pour le nouvel arrivant dans le monde de Django. Par ailleurs, il est des cas spécifiques où les vues-classes ne conviendraient tout simplement pas. Enfin, comme vous allez le voir, ce que vous avez appris au fil de ce livre avec les vues-fonctions va vous être extrêmement utile pour les vues-classes.

Vues-classes vs vues-fonctions

Pourquoi privilégier les vues-classes plutôt que les vues-fonctions ? La première réponse est théorique. Django fonctionne sur le principe des classes un peu partout dans le framework : les modèles, les formulaires, l'`admin`, etc., il est donc logique que les vues soient elles aussi des classes. Ensuite, les vues-classes permettent d'écrire moins de code puisque vous profitez de l'héritage des classes ; c'est toujours une raison suffisante ! Enfin, les vues-classes aident également à écrire du code mieux structuré. Il vous sera, par exemple, aisé de créer un mixin en charge de générer du JSON. Par la suite, toutes les vues qui auront besoin de cette fonctionnalité en hériteront.

Au bout de quelques temps, vous aurez certainement à votre disposition quelques modules réutilisables dans vos différents projets pour gérer de nombreuses situations : HTML, JSON…

Les verbes REST et les vues-classes

Les verbes REST sont gérés de manière particulièrement intelligente dans les vues-classes. Quand vous étiez obligé d'écrire :

```
def ma_vue(request) :
    if request.method == 'POST'
        # Faire quelque chose
    elif request.method == 'GET'
```

vous pouvez maintenant écrire simplement :

```
from django.views.generic.base import View
class MaVue(Views) :
def get(self, request, **kwargs) :
    # Faire quelque chose
def post(self, request, **kwargs) :
   # Faire autre chose
```

C'est à la fois simple et concis et cela évite d'aller chercher dans les entrailles de l'objet `request`.

Django vous propose les méthodes `HEAD`, `POST`, `GET`, `DELETE` et `PUT`, ce qui vous permet de générer une fonction par méthode et rend la création d'API REST d'une simplicité déconcertante.

Les héritages de vues

Au cours du développement de votre application, vous vous apercevrez souvent que certaines de vos vues font plus ou moins la même chose : soit le template change, soit la queryset n'est pas la même ou bien le modèle est différent.

Dans le cadre de vues-fonctions, il est possible de créer des fonctions limitant cette duplication de code, mais ce n'est pas très simple à maintenir ; vous aimeriez parfois englober un ensemble de fonctions dans un objet pour l'identifier plus facilement… Et cela, les vues-fonctions ne peuvent pas vous l'offrir. En revanche, avec les vues-classes, il est possible de proposer un ensemble de fonctionnalités sous le même objet. Dans le cas d'applications réutilisables, c'est même encore plus puissant : offrez vos outils sous forme de classes que vos autres applications pourront simplement réutiliser en en héritant. Par exemple :

```
class JsonOrHtmlView(DetailView) :
    def json_render_to_response(self,
                      context,
                     **response_kwargs):
```

```
        """
        Renvoie une réponse au format JSON
        """
        self.response_class = HttpResponse
        response_kwargs['content_type'] = 'application/json'
        return self.response_class(
            self.object.to_json(),
            **response_kwargs
        )
    def html_render_to_response(self,
                    context,
                    **response_kwargs):

        response_kwargs['content_type'] = 'text/html'
        return super(DetailView, self).render_to_response(
            context,
            **response_kwargs)

    def render_to_response(self, context):
        if self.request.is_ajax():
            return self.json_render_to_response(context)
        else:
            return self.html_render_to_response(context)
```

Cette classe possède trois méthodes : `json_render_to_response`, `html_render_to_response` et `render_to_response`. Quand vous l'utiliserez, elle sera en mesure de renvoyer indifféremment du JSON ou du HTML, en fonction des entêtes HTTP de la requête. Ainsi, elle sera dans tous vos projets, sous réserve bien entendu que vous ayez créé une méthode `to_json` sur vos modèles, qui devra renvoyer la représentation de l'objet au format JSON.

N'hésitez pas à modifier cette classe pour la faire correspondre à vos besoins et, surtout, écrivez les vôtres. C'est une occasion unique de garder précieusement les classes les plus utiles que vous aurez créées dans un module importable directement dans vos nouveaux projets, et ainsi de gagner un temps précieux.

Du bon usage des vues-classes

Quand on prend en main un nouveau framework web, on constate que tout est pensé au niveau de la ressource, ce qui est assez logique en termes de conception. En revanche, au moment où l'on met en place le projet, on s'aperçoit souvent que ce n'est pas notre besoin principal et l'on va bien souvent devoir présenter à l'utilisateur un ensemble d'objets disparates.

Prenons l'exemple d'un journal en ligne. Bien entendu, le corps de la page correspond à un article unique. Son titre, son auteur, son contenu, éventuellement la photo

d'illustration, tout cela est parfait. Seulement, tout se complique quand le chef de projet vous demande de présenter une barre latérale avec les derniers articles ou les derniers commentaires… Cette fois, vous ne pouvez plus utiliser toutes les aides de votre framework et vous êtes obligé de mettre en place une solution « maison » qui sera souvent difficile à maintenir, qui vous obligera à « tordre » le framework pour le faire correspondre à vos besoins.

Les vues-classes, dans cette perspective, offrent une véritable solution. Elles faciliteront la mise en place d'une architecture à plusieurs niveaux. Le premier niveau, le plus bas, va correspondre à une sorte d'API de vos objets. Vous implémenterez aisément, à l'aide des vues génériques, les classes listes et détails de vos objets. Dans le second niveau, vous allez implémenter des vues qui hériteront des premières, mais en modifiant la méthode `get_context` pour qu'elle aille remplir le contexte avec les autres objets dont vous avez besoin. Cette méthode utilisera elle aussi les vues que vous avez définies au premier niveau de votre API. Si vous le souhaitez, ou si votre application devient vraiment complexe, vous implémenterez un troisième niveau qui surclassera encore ce second ensemble. À la fin, vous n'aurez donc plus qu'à appeler le second ou le troisième niveau de votre architecture. Votre API restera la pierre angulaire de votre organisation et vous aurez rempli votre objectif : proposer une architecture à la fois modulaire et extensible renvoyant des collections d'objets hétéroclites.

Pour mieux vous expliquer cette problématique à laquelle vous serez confronté tôt où tard, nous vous proposons un petit exemple. Il s'agit du journal en ligne d'une école, très basique mais suffisant pour que vous puissiez en comprendre l'architecture. Voici tout d'abord les modèles :

```
class Section(models.Model) :
    nom = models.CharField(max_length = 250)
    description = models.TextField()

class Categorie(models.Model) :
   nom = models.CharField(max_length=250)

class Auteur(models.Model) :
    nom = models.CharField(max_length = 250)
    section = models.ForeignKey(Section)

class Article(models.Model) :
    auteur = models.ForeignKey(Auteur)
    categories = models.ManyToManyField(Categorie)
    titre = models.CharField(max_length=250)
    chapo = models.TextField()
    contenu = models.TextField()
    date = models.DateField()
```

Vous avez donc des sections, qui correspondent aux spécialités des élèves, des catégories qui définissent de quoi parlent les articles, des auteurs et enfin des articles.

La première étape va donc consister à créer le premier niveau d'API dans vos vues. Dans cet exemple très simple, vous pourriez passer directement à l'étape suivante. Cependant, comme il est fort probable que vous deviez tôt ou tard modifier ce niveau, il est plus simple de pouvoir y avoir directement accès dès maintenant. L'évolution de votre application n'en sera que plus simple.

```
From django.views.generic.list_detail import ListView, DetailView
Class SectionList(ListView) :
    model = Section
class SectionDetail(DetailView) :
    model = Section

class CategorieList(ListView) :
    model = Categorie
class CategorieDetail(DetailView) :
    model = Categorie

class AuteurList(LIstView) :
    model = Auteur
class AuteurDetail(DetailView) :
    model = Auteur

class ArticleList(ListView) :
    model = Article
class ArticleDetail(DetailView) :
    model = Article
```

Cette première étape est très succincte. Pour le moment, nous ne définissons rien de particulier dans les vues. Cependant, nous nous réservons la possibilité de facilement les modifier si nous en ressentons le besoin au cours du développement. Ensuite, nous devons écrire le second niveau de notre application. Sur la page listant toutes les catégories, nous voulons proposer un encart qui présente les auteurs les plus prolifiques.

La première idée qui vient est donc de créer une classe `AuteurProlifique` dont la queryset serait `Auteur.objects.all().order_by(num_article)[:10]`, par exemple, pour récupérer les dix auteurs les plus prolifiques. Petit problème cependant, le nombre d'articles d'un auteur n'existe pas dans vos modèles. Il n'est pas possible de créer une méthode qui le calcule, car vous ne pouvez pas faire un `order_by` sur une méthode, mais uniquement sur un champ. En effet, cette méthode utilise des mécanismes de base de données beaucoup plus rapides et efficients qu'un code Python. Pour sortir de cette impasse, il faut procéder en deux temps. Premièrement, nous devons rédiger une méthode du modèle `Auteur` qui va renvoyer le nombre d'articles qu'un auteur a

émis (un simple `self.articles.count()`). Ensuite, il faut créer un nouveau champ `num_article` sur le modèle `Auteur`. Enfin, nous devons modifier la méthode `save` du modèle `Article` pour qu'une sauvegarde d'un article modifie la valeur de `num_articles` de son auteur :

```
class Auteur(models.models) :
    nom = models.CharField(max_length = 250)
    section = models.ForeignKey(Section)
    num_articles = models.IntegerField(max_length=4)

    def get_articles_count(self) :
        return self.articles.count()

class Article(models.Model) :
    auteur = models.ForeignKey(Auteur)
    categories = models.ManyToManyField(Categorie)
    titre = models.CharField(max_length=250)
    chapo = models.TextField()

    def save(self, *args, **kwargs) :
        super(self, Article).save(*args, **kwargs)
        self.auteur.num_articles = self.auteur.get_num_articles()
        self.auteur.save()
```

Après ces quelques modifications au niveau du modèle, nous avons un champ `num_articles` sur le modèle `Auteur`. Nous allons donc pouvoir créer la classe `AuteurProlifique` :

```
class AuteurProlifique(ListView) :
    def get_queryset(count) :
        return Auteur.objects.all().order_by('-num_article')[:count]
```

Pour ne pas confondre les vues qui exposent des fonctionnalités et celles qui vont être appelées par les URL, nous allons créer un nouveau fichier de vues et changer les appellations. Par exemple :

```
class ListCategories(CategorieList) :
    def get_context_data(self) :
        context = self.context
        context['auteurs'] = AuteurProlifique(count=5).get_context_data()
```

Grâce à ces quelques lignes, il est désormais très simple de renvoyer la liste des catégories ainsi que celle des auteurs les plus prolifiques.

En procédant de cette manière, nous pouvons toujours renvoyer la liste des catégories, avec ou sans les auteurs prolifiques, et aussi modifier très simplement le nombre

d'auteurs à renvoyer. Il est enfin tout à fait possible d'utiliser la classe `AuteurProlifique` d'une autre classe, soit seule, soit dans un autre contexte, sans duplication de code bien entendu.

Utiliser les capacités d'agrégation de Django

La première méthode que vous venez de voir pour retourner la queryset de `AuteurProlifique` serait beaucoup plus simple en utilisant les capacités d'agrégation de Django. Les concepts d'agrégation font usage de mécanismes très rapides fournis par la base de données. Pour cet exemple, vous n'avez pas besoin d'ajouter un champ `num_articles` à votre modèle ; il suffit d'importer `Count` de `django.db.models` :

```
from django.db.models import Count
```

et d'écrire votre méthode `get_queryset` de la façon suivante :

```
class AuteurProlifique(ListView) :
    def get_queryset(count) :
        return
Auteur.objects.annotate(num_article=Count('articles')).order_by('-
num_article')[:count]
```

Dans cette configuration, vous vous trouverez souvent dans une architecture à trois niveaux : tout d'abord, les classes simples comme `CategorieList`, puis des classes utilitaires comme `AuteurProlifique` et enfin des classes destinées à être appelées dans vos vues comme `ListCategorie`.

12

L'affichage de l'information : les templates

L'affichage de l'information est l'un des points les plus importants d'un framework. Le système doit être suffisamment souple pour réaliser des affichages complexes, mais aussi facile à prendre en main par un intégrateur qui ne connaît pas Python ou ses subtilités. Découvrons dans ce chapitre la solution que Django propose au travers de son langage de templates.

Le langage de templates

Le langage de templates de Django se veut à la fois simple et puissant : simple pour qu'un nouveau venu dans le monde de Django puisse le prendre en main en quelques minutes, et puissant pour que les besoins d'affichage complexe puissent être satisfaits. Avant de détailler le fonctionnement de ce langage, il est nécessaire de revenir sur quelques points fondamentaux.

Le template et le contexte

Le template et le contexte sont les deux éléments qui vont permettre d'afficher des données formatées à l'utilisateur.

Afficher une variable

L'affichage d'un template est toujours la résultante d'une compilation de deux éléments : le template et le contexte. Le premier est un fichier texte, très souvent du HTML, et le second est un dictionnaire Python. Chaque élément du dictionnaire va être appelé au sein du template à l'aide de mots-clés spécifiques au langage de templates. Lors de la compilation, chaque mot-clé est remplacé par la valeur qui lui correspond dans le dictionnaire.

Prenons pour exemple le template extrêmement simple qui suit :

```
Bonjour, {{ utilisateur }}
```

Utilisons le dictionnaire suivant :

```
{'utilisateur': 'Jean'}
```

Le résultat de la compilation sera :

```
Bonjour, Jean
```

En revanche, avec le dictionnaire :

```
{'utilisateur' : 'Pierre'}
```

le résultat sera :

```
Bonjour, Pierre
```

En revanche, avec le dictionnaire `{'age': 18}`, le résultat sera :

```
Bonjour,
```

En effet, ce dernier dictionnaire ne contenant pas la clé `utilisateur`, le moteur de templates laisse le champ vide.

Vous venez de voir deux exemples de compilation du même template. Ce qui change, c'est le contexte. La première façon de le charger dans le template est donc d'utiliser les doubles accolades : `{{ clé de dictionnaire }}`.

Ceci étant, il vous faudra souvent accéder à des attributs de l'objet Python que vous manipulez. Django utilise la syntaxe classique du « point » pour y accéder. Prenons l'exemple d'une classe `Utilisateur` que l'on peut définir comme ceci :

```
class Utilisateur(object):
    def __init__(self, nom, prenom, age):
        self.nom = nom
        self.prenom = prenom
        self.age = age
```

Si vous avez un objet `utilisateur` dans votre contexte et souhaitez afficher son nom, vous pouvez utiliser le point :

```
{{ utilisateur.nom }}
```

Afficher une série de variables

Il est des cas où ce que vous souhaitez afficher ne se limite pas à un seul objet. Les doubles accolades ne suffisent alors plus : vous aurez besoin d'une boucle.

Imaginons la liste d'objets suivante :

```
liste = ['Pierre', 'Jean', 'Philippe']
```

Vous l'utiliserez dans un contexte, c'est-à-dire un dictionnaire :

```
{'utilisateurs' : liste}
```

Dans votre template, vous allez donc écrire une boucle :

```
{% for utilisateur in utilisateurs %}
{{utilisateur}}
{% endfor %}
```

Cette boucle très simple signifie : « Pour chaque `utilisateur` dans la liste `utilisateurs`, afficher la valeur correspondante du dictionnaire. » Ce code va donc renvoyer les trois utilisateurs les uns à la suite des autres.

Par des jeux d'écriture, Django fait la différence entre les variables qui doivent être affichées (doubles accolades) et les ordres logiques qui doivent être évalués (une seule accolade suivie du signe %).

Branchements logiques

L'autre cas très courant consiste à n'afficher une variable que si elle passe un test. Pour reprendre notre exemple sur les utilisateurs, imaginons que vous ne voulez afficher le nom que s'il s'agit de Pierre.

Dans ce cas, il faut écrire :

```
{% if utilisateur == 'Pierre' %}
{{ utilisateur }}
{% endif %}
```

ou, plus simple :

```
{% ifequal utilisateur 'Pierre' %}
{{ utilisateur }}
{% endifequal%}
```

Dans le cas où vous voulez exécuter une autre série d'instructions quand la condition n'est pas réalisée, la syntaxe se complète comme suit :

```
{% if utilisateur == 'Pierre' %}
{{ utilisateur }}
{% else %}
Bonjour
{% endif %}
```

ou :

```
{% ifequal utilisateur 'Pierre' %}
{{ utilisateur }}
{% else %}
Bonjour
{% endifequal %}
```

Ici, l'utilisateur sera affiché si son nom est Pierre. Sinon, `Bonjour` sera présenté.

Les différents signes logiques que vous pouvez utiliser dans vos conditions sont les suivants :

- `==` : égal à ;
- `!=` : différent de ;
- `<` : inférieur à ;
- `>` : supérieur à ;
- `<=` : inférieur ou égal à ;
- `>=` : supérieur ou égal à ;
- `in` : pour que la condition soit vraie, il faut que l'élément de gauche se trouve dans celui de droite. Le plus souvent, il s'agira de vérifier si un élément se trouve bien dans une liste ;
- `not in` : pour que la condition soit vraie, il faut que le terme de gauche ne se trouve pas dans celui de droite. Le plus souvent, il s'agira d'une liste.

En langage Django, les morceaux de code entre les signes {% et %} sont appelés *template tags*. Ils ont la particularité d'être évalués par Django, mais ils ne sont pas affichés dans le rendu final.

Les filtres

D'une utilisation très similaire à celle des template tags, les filtres font un peu plus qu'afficher ou non une information. Ils la modifient pour la faire correspondre à vos besoins. Ils récupèrent automatiquement la variable à modifier en utilisant le signe |. Par exemple, le filtre `upper` affiche une variable du contexte en majuscules :

```
{{variable|upper}}
```

Le filtre date

Certains filtres sont plus complexes et ont besoin de paramètres pour modifier correctement la variable. C'est le cas de `date`, typique des filtres requérant des arguments. Sa fonction est de formater une date. Par exemple, pour afficher une date au format « Jour Mois Année », vous devrez écrire :

```
{{votre_date|date : 'j n Y' }}
```

La syntaxe est donc la suivante : votre variable, le signe |, le filtre à utiliser, le signe : et, entre guillemets, les paramètres à utiliser. Ici, il est question des paramètres de formatage de date que vous trouverez dans la documentation en ligne de Django (https://docs.djangoproject.com/en/dev/ref/templates/builtins/?from=olddocs#date).

Revue de code

Bien qu'incomplet, ce premier tour d'horizon vous donne les clés qui vous aideront à comprendre la suite de ce chapitre. Il existe en effet de nombreux autres template tags et filtres que Django propose par défaut, que vous trouverez dans des projets tiers ou que vous écrirez vous-même. En définitive, les templates se composent de variables, de template tags et de filtres.

L'organisation des templates

L'organisation des templates doit être traitée le plus tôt possible lors de la création de votre projet. En effet, il est très facile de se laisser déborder et de ne plus être en mesure de gérer convenablement la multitude de vues et de templates. C'est pour-

quoi, nous vous proposons de suivre une méthode simple qui vous permettra de répondre à tous les cas ou presque.

Un dossier template pour le projet

Créez un dossier `template` à la racine de votre projet. La tentation est forte d'en faire l'unique ressource de templates, pourtant nous vous conseillons de résister ! En effet, si tous vos templates sont dans le même dossier, vous perdez du même coup le découplage de vos applications. C'est pourquoi, il est conseillé de placer dans le dossier `template` un fichier `base.html` qui contiendra les blocs les plus importants de votre projet. Il s'agira d'une sorte d'échafaudage, aussi grossier que possible, qui va simplement rendre disponible la ou les feuilles de style du projet et une série de blocs de base. L'idée ici est bien entendu de profiter de l'héritage des templates que vous offre Django. Par exemple :

```
<!DOCTYPE html>
<html>
  <head>
    {% block head %}
    {% endblock head %}
  </head>
  <body>
    {% block header %}
    {% endblock header %}
    {% block menu %}
    {% endblock menu %}
    {% block content %}
    {% endblock content %}
    {% block footer %}
    {% endblock footer %}
  </body>
</html>
```

Bien entendu, cet exemple est particulièrement simpliste. Dans le cas d'un vrai projet, vous aurez sans doute des blocs différents et peut-être aurez-vous déjà défini certains éléments. Néanmoins, ce template de base vous permet de bien voir le principe et la structure de ce type de page.

Un dossier template par application

Comme chacune de vos applications va proposer un ensemble de vues, donc un ensemble de templates, il vous faut créer un dossier `template` par application. Afin, là aussi, de rassembler toutes les informations utiles communes à toutes les pages de votre application, créez un fichier `base.html` qui héritera du fichier `base.html` à la

racine de votre projet. Ensuite, créez un dossier portant le nom de votre application et contenant les templates relatifs uniquement à cette dernière.

Prenons l'exemple d'une application qui s'appelle `snippets`. Vous aurez l'architecture suivante :

```
/projet/templates/base.html
/projet/snippets/templates/base.html
/projet/snippets/templates/snippets/all_snippets.html
```

Les dossiers includes

Très rapidement, vous vous apercevrez que certaines parties de vos templates se répètent de page en page. Il est alors utile de créer un dossier `includes` dans chacune de vos applications afin d'alléger vos templates et de les rendre plus DRY. Lorsque vous souhaiterez modifier l'affichage du menu par exemple, toutes les pages avec `includes menu.html` bénéficieront du changement. Enfin, certains éléments sont utilisés un peu partout dans votre projet, comme l'en-tête ou le pied de page ; ils peuvent tout à fait être intégrés dans le dossier `template` à la racine de votre application, dans un dossier `includes` spécifique.

Dans tous les cas, évitez autant que possible les références croisées. Quand l'une de vos applications emploie un `includes` d'une autre application, c'est que vous avez un problème. Par exemple, peut-être que cette portion de code aurait plus sa place dans le dossier `includes` général. Pour reprendre l'exemple de l'application `snippets`, vous pouvez définir un dossier `includes` général de la façon suivante :

```
/projet/snippets/includes/<un fichier include>
```

Vous restez ainsi totalement découplé.

Créer des template tags et des filtres

Vous avez vu combien il était aisé d'utiliser les template tags que propose Django dans les templates. Cependant, même si la collection de filtres et de tags de Django est très importante, il arrive tôt ou tard un moment où le template tag que vous cherchez n'existe pas. Heureusement, Django vous offre la possibilité de créer les vôtres.

Les filtres : modifier une variable

Vous utiliserez les filtres lorsque vous souhaitez faire une opération sur un élément du `Context`. Ils prennent en argument une variable que vous souhaitez modifier et, parfois, un paramètre supplémentaire. Dans votre template, voici comment vous les emploierez : `{variable:filtre : 'argument'}`.

Dans un premier temps, nous allons créer un filtre très simple, qui passe en gras le texte que vous lui donnez en argument à l'aide des balises HTML `<b></b>`. Nous le nommerons `bold` (gras en français). Il se présentera comme suit dans votre template :

```
{variable:bold}
```

Ce premier filtre ne possède donc pas de paramètre supplémentaire.

Structure de fichier

Avant de réellement écrire votre filtre, il faut préparer le travail. Pour cela, et afin que votre filtre soit disponible dans les templates, il faut que vous suiviez une arborescence particulière.

Dans l'une de vos applications, vous devez créer un dossier `templatetags`. Dans celui-ci, écrivez un fichier `__init__.py` vide, qui indique à Django que le dossier est un paquet qu'il pourra utiliser. Enfin, au sein de ce dossier, créez un fichier que vous pouvez nommer comme bon vous semble et dans lequel vous écrirez vos filtres. Bien entendu, vous pouvez écrire plusieurs fichiers si vous souhaitez ne rendre disponibles que certains filtres spécifiques en fonction du template employé.

Avant d'écrire vos filtres, vérifiez bien que les applications dans lesquelles vous les avez placés sont bien enregistrées dans les `INSTALLED_APPS` de votre fichier `settings.py`. Sinon, Django ne trouvera pas vos filtres.

Pour l'exemple, nous allons créer le fichier `extra_filtre.py` dans le dossier `templatetags`. Et c'est tout naturellement dans ce fichier que nous allons écrire notre premier filtre.

Écriture d'un premier filtre

Avant de commencer, il faut savoir que Django tient à jour une listes des tags et des filtres utilisables. Elle se trouve dans une variable nommée `register` qui elle-même est une instance de l'objet `template.Library`. Donc, avant de commencer, nous devons charger cette bibliothèque grâce aux deux lignes de code suivantes :

```
from django import template
register = template.Library
```

Maintenant, écrivez votre filtre :

```
def bold(value) :
    return '<b>%s</b>'%value
```

Le code du filtre est très simple. Il faut maintenant l'enregistrer dans la bibliothèque des tags et filtres de Django :

```
register.filter('bold', bold)
```

Cette dernière ligne indique à Django d'enregistrer un nouveau filtre du nom de `bold` qui exécutera la fonction `bold`.

Dans l'un de vos templates, vous pouvez charger votre fichier `extra_filtre` puis utilisez le filtre :

```
{% load extra_filtre %}
{{'un test'| bold}}
```

Lorsque vous visiterez votre template, vous aurez certainement une mauvaise surprise. En effet, le filtre a certes bien fonctionné, mais pas comme on le souhaitait. Vous devez voir s'afficher à l'écran :

```
<b>un test</b>
```

Et finalement c'est bien ce que vous avez demandé, non ? Seulement, vous auriez souhaité que `<b>` soit interprété par le navigateur et non affiché. Comme vous vous en doutez certainement, tous les caractères spécifiques au HTML sont échappés, c'est-à-dire convertis dans leur notation HTML. Ainsi, le signe < est converti en `;&lt`. C'est une mesure de sécurité qui évite qu'un utilisateur mal intentionné puisse injecter du HTML dans vos pages. Toutefois, dans le cas présent, c'est vous qui souhaitez y intégrer du HTML ; il n'y a donc pas de raison pour que l'on ne vous laisse pas faire. Pour cela, il faut utiliser `mark_safe`, qui indique à Django de traiter la variable comme du HTML. Reprenons donc le filtre et modifions-le comme suit :

```
from django.safestring import mark_safe
def bold(value):
    return mark_safe('<b>%s</b>'%value)
```

Rechargeons la page : le texte apparaît bien en gras.

Les filtres avec argument

Maintenant que vous maîtrisez le principe général, il est temps de se pencher sur les filtres avec arguments. Nous allons prendre pour exemple l'opérateur mathématique « multiplier ». Il se présente comme suit :

```
def mult(value, arg) :
    return int(value) * int(arg)
template.register('mult', mult)
```

Vous l'utiliserez dans votre template de la manière suivante :

```
{{'12'|mult :'5'}}
```

Les tags : agir au niveau du contexte

Les tags sont d'un usage un peu plus complexe. En effet, alors que les filtres ne font que modifier une variable, les tags doivent utiliser une logique plus complexe et agir directement au niveau du contexte, c'est-à-dire avant que la compilation du template ait lieu.

Revenons rapidement sur la compilation des templates. Pour Django, un template est une suite de nœuds qui sont des instances de la classe `django.template.Node`. Chacun est pourvu d'une méthode `render` qui indique à Django comment ce nœud doit être compilé. Lorsque vous créez un template tag, vous devez donc indiquer à Django quel est ce nouveau nœud et comment le compiler, c'est-à-dire écrire la méthode `render` de ce nœud.

Avant d'écrire votre premier tag, voyons comment l'utiliser dans votre template. Un bon point de départ pourrait être le suivant :

```
{% nom_du_tag 'arguments' %}
```

Une fonction de template tag peut prendre deux arguments : `parser` et `token`. Le premier agit sur ce qui se trouve entre deux tags, par exemple entre `if` et `endif` ou entre `for` et `endfor` ; vous ne l'utiliserez pas dans ce premier exemple. Le second correspond à ce qui se trouve entre `{%` et `%}`, en l'occurrence `nom_du_tag` et `arguments`. Cependant, pour Django, tout ceci n'est qu'un seul bloc. Il faut donc faire en sorte de diviser ces deux objets dans deux variables distinctes, ce que permet `token.split_content()`. Voici donc à quoi peut ressembler votre premier template tag :

```
def do_nom_du_tag(parser, token) :
    tag_name, args = token.split_content(token)
```

Vous voilà avec un `tag_name` et un `args`. Il faut maintenant créer un nœud que Django va pouvoir compiler :

```
class MonTagNode(template.Node) :
    def __init__(self, string) :
        self.string = string
    def render(self) :
        return self.string
```

Ce nœud doit être instancié par la fonction `do_nom_du_tag` :

```
def do_nom_du_tag(parser, token) :
    tag_name, args = token.split_content(token)
    return MonTagNode(args)
```

Puis vous devez enregistrer votre tag pour le rendre fonctionnel :

```
register.tag('nom_du_tag', do_nom_du_tag)
```

Voici maintenant un exemple concret et complet : vous souhaitez créer un tag qui écrit `Bonjour <nom de l'utilisateur>`, le nom d'utilisateur étant bien entendu donné en argument. Ce template pourra être écrit comme suit :

```
{% bonjour 'Jean' %}
```

ou, si vous utilisez un modèle `User` :

```
{% bonjour user .username %}
```

Votre classe `Node` pourra être écrite comme suit :

```
class HelloNode(template.Node) :
    def __init__(self, user) :
        self.user = user
    def render(self) :
        return 'Bonjour %s' % self.user
```

et votre fonction `salutation` :

```
def salutation(parser, token) :
    tag_name, user = token.split_contents()
    return HelloNode(user)
register.tag('bonjour', salutation)
```

Les décorateurs

Comme il peut être fastidieux d'utiliser la syntaxe `register.tag('bonjour', salutation)`, remplacez-la par son décorateur, de la manière suivante :

```
@register.tag(name='bonjour')
def salutation(parser, token) :
    tag_name, user = token.split_contents()
    return HelloNode(user)
```

Cette utilisation des décorateurs s'applique également pour les filtres :

```
@register.filter
def mult(value, arg) :
    return int(value) * int(arg)
```

Ajouter des arguments aux tags

Dans les deux exemples précédents, vous n'avez pas employé d'arguments supplémentaires. Pourtant, vous avez la possibilité de le faire. Par exemple, si vous souhaitez afficher un message personnalisé en fonction du genre de l'utilisateur, vos template tags au niveau du template ressembleraient à ce qui suit :

```
{% bonjour user.username 'mr' %}
```

et

```
{% bonjour user.username 'mme' %}
```

Ici, il faudrait que le tag `{% bonjour user.username 'mr' %}` renvoie `Bonjour Monsieur <nom de l'utilisateur>` et que `{% bonjour user.username 'mme' %}` renvoie `Bonjour Madame <nom de l'utilisatrice>`.

La méthode `token.split_content()` va dans ce cas vous renvoyer trois valeurs. Votre fonction `salutation` devient donc :

```
@register.tag(name='bonjour')
def salutation(parser, token) :
    tag_name, user, genre = token.split_contents()
    return HelloNode(user, genre)
```

Comme le genre est maintenant renvoyé à `HelloNode`, il faut lui indiquer ce qu'il doit en faire :

```
class HelloNode(template.Node) :
    def __init__(self, user, genre) :
        self.user = user
        self.genre = genre
    def render(self) :
        if self.genre == 'mr' :
            genre = 'Monsieur'
        elif self.genre == 'mme' :
            genre = 'Madame'
        return 'Bonjour %s %s' % (genre, self.user)
```

Si ces exemples sont très simples, pour des raisons didactiques, vous pouvez aisément en faire ce que bon vous semble. L'idée ici est que vous puissiez comprendre les fondements du fonctionnement des tags dans Django avant d'aller plus loin. Vous utiliserez assez peu souvent ce type d'écriture. En effet, Django vous propose d'écrire des template tags de façon beaucoup plus simple en fonction du résultat que vous souhaitez obtenir, avec les décorateurs.

Les tags, contrairement aux filtres, ne sont pas affichés par Django mais évalués. Il sont reconnaissables à leur écriture `{% tag %}`, comme dans `{% csrf_token %}`. Vous trouverez aussi parfois l'écriture avec variable `{% tag variable %}`, comme dans `{% if variable == 'test' %}{% endif %}`. Ces différentes écritures correspondent à des cas d'utilisation distincts. Les exemples que vous venez d'implémenter entrent dans la catégorie des `simple_tag`, qui prennent une chaîne de caractères en argument et renvoie également une chaîne de caractères. En utilisant le template tag `simple_tag`, vous n'avez plus besoin de faire appel à une classe de type `Node`. Vous pouvez simplement écrire :

```
@register.simple_tag(name='bonjour')
def salutation(user, genre) :
    if self.genre == 'mr' :
        genre = 'Monsieur'
    elif self.genre == 'mlle' :
        genre = 'Mademoiselle'
    return 'Bonjour %s %s' % (genre, user)
```

Vous en conviendrez, c'est bien plus simple. Pourtant, parfois, il y a des cas où vos besoins seront différents. Si vous souhaitez agir directement sur le contexte, en l'enrichissant de nouvelles données, vous devrez cette fois utiliser le décorateur `inclusion_tag`. Par exemple, pour afficher une liste des derniers articles, vous écrirez tout d'abord une fonction récupérant ces derniers dans la base de données :

```
def derniers_articles() :
    articles = Articles.objects.all().order_by('-date')[:5]
    return {'derniers_articles' : articles}
```

Ici, vous avez donc récupéré les cinq derniers articles et vous les retournez. Bien entendu, vous avez besoin d'un template pour les afficher :

```
<ul>
{% for articles in derniers_articles %}
<li>{{article.titre}}</li>
{% endfor %}
</ul>
```

Ce template va afficher une liste des titres des cinq derniers articles. Il faut maintenant expliquer à votre fonction `derniers_articles` qu'il faut utiliser ce template pour le rendu :

```
@register.inclusion_tag('derniers_articles.html')
def derniers_articles() :
    articles = Articles.objects.all().order_by('-date')[:5]
    return {'derniers_articles' : articles}
```

Vous pouvez maintenant écrire dans votre template :

```
{% derniers_articles %}
```

ce qui affichera, sous forme de liste, les derniers articles de votre base de données.

Conclusion

Les filtres et les template tags sont souvent très utiles, car ils simplifient l'écriture des templates. En revanche, il faut faire attention à ne pas en abuser, en particulier en ce qui concerne les `inclusion_tag` qui font des requêtes en base de données. Ils sont en effet plus complexes à déboguer que les éléments de contexte présents directement dans la vue. Le programmeur devra en effet consulter le template, puis les fichiers `templatetags.py` pour enfin trouver le template tag qui produit le contenu qu'il n'aura pas trouvé dans sa vue. Néanmoins, il est des cas où cet inconvénient est directement contrebalancé par l'avantage en termes de clarté et de simplicité dans le template. C'est dans ce type de situation que l'usage des template tags est souhaitable.

13

Dialogue avec l'utilisateur : les formulaires

Depuis les débuts du Web, les formulaires sont un élément central des interfaces avec l'utilisateur. En effet, ils sont la principale méthode mise à votre disposition pour dialoguer avec votre visiteur. Le framework Django leur accorde une place importante et vous permet de créer facilement des formulaires réutilisables et dynamiques.

L'objet Form

Toute l'implémentation des formulaires de Django se fait autour de l'objet `Form`, qui peut être instancié, manipulé, affiché dans le template et traité au niveau de vos vues. Comme il s'agit d'un objet, vous bénéficiez de tous les avantages de l'héritage et de la possibilité de le traiter comme un tout.

L'objet `Form` repose sur deux concepts importants.

- Un *field* est un champ de formulaire. C'est également une instance de classe, de la même manière que les champs de `Model`. L'une de ses méthodes obligatoires concerne le processus de validation.
- Les widgets sont l'une des propriétés du field. Il s'agit de la manière dont ce champ est représenté au niveau du template. Il peut s'agir d'un champ de type texte, d'un bouton radio, d'une suite de choix à cocher, etc.

Django a défini pour vous un ensemble de fields qui vous serviront dans vos propres formulaires. Cet ensemble correspond bien entendu aux champs de modèle car, très souvent, vous souhaiterez représenter des objets de vos modèles via un formulaire. Par ailleurs, Django permet même de générer ces formulaires automatiquement grâce à la classe `ModelForm` dont nous reparlerons plus loin. Chacun de ces champs de formulaire possède un système de validation spécifique pour vérifier l'exactitude des données entrées par l'utilisateur.

Enfin, bien entendu, vous pourrez employer vos propres classes de champs avec vos propres systèmes de validation, ainsi que l'héritage de classe pour créer des fields à partir de champs existants, modifier les widgets normalement utilisés… En bref, les formulaires dans Django sont un sujet presque aussi vaste que les modèles, dont ils s'inspirent beaucoup. Vous verrez également que les vues génériques ne sont pas non plus en reste quand il s'agit d'écrire automatiquement des formulaires, pour la création et la modification des objets par exemple.

Un premier exemple : formulaire d'enregistrement à une newsletter

Avant d'entrer plus en détail dans les arcanes de la création de formulaire, découvrons-en un premier exemple. Nous vous proposons un formulaire d'enregistrement à une newsletter. Pour s'y abonner, le visiteur doit simplement transmettre son e-mail. Le formulaire peut donc être représenté comme suit :

```
from django import forms
class Abonnement(forms.Form) :
    utilisateur = forms.EmailField()
```

Comme vous le voyez, la nouvelle classe que vous venez d'écrire est fort simple.

1. Vous importez `forms` depuis Django. Cela vous donne accès à l'ensemble des classes et fonctions liées aux formulaires.
2. Vous créez votre classe héritant de `Form`.
3. Vous définissez un champ de type `EmailField`.

Avec ces quelques informations, vous avez déjà un formulaire capable de beaucoup de choses. Par exemple, le simple fait de définir un champ de type `EmailField` vous permet de bénéficier gratuitement de la validation automatique du champ `email`.

Traitement du formulaire

Pour que ce formulaire soit présenté à l'utilisateur, il faut l'instancier dans un contexte. Par exemple, dans l'une de vos vues, vous pouvez écrire ce qui suit.

```
from django import render
def souscription(request):
    form = Abonnement()
    if request.method == 'POST':
        form = Abonnement(request.POST)
        if form.is_valid():
            email = form.cleaned_data['email']
            # Faites ce que vous voulez avec l'e-mail validé.
        else
            # Renvoyez le formulaire avec les erreurs.
    else:
        return render(request,'abonnement.html', {'form': form})
```

Cette fonction va se comporter différemment suivant que la requête soit appelée via la méthode `POST` ou non. Dans le cas où la requête est de type `POST`, vous instanciez votre objet `Form` avec les données présentes dans le formulaire et le renvoyez à la vue.

L'appel à la méthode `form.is_valid` sert à valider le formulaire. Cela se fait en deux temps, mais c'est tout à fait transparent pour vous : la méthode `is_valid` valide d'abord chaque champ, puis le formulaire dans son ensemble. Dans le cas présent, cela consiste à valider le champ `EmailField`. La validation automatiquement proposée par Django sur ce type de champ vérifie que la valeur de ce dernier est bien de la forme `<identifiant>@<nom de domaine>.<extension>`. Dans cet exemple, il n'y a pas de validation du formulaire dans son ensemble, car il n'y a qu'un seul champ. En revanche, dans un formulaire plus complet où vous souhaiteriez vérifier en plus un champ « mot de passe », cette validation globale aurait lieu.

Si le formulaire est valide, vous avez maintenant accès à `form.cleaned_data`, qui est un dictionnaire des valeurs des champs validés. Vous utiliserez ce dictionnaire pour traiter les données utilisateur comme il se doit et, par exemple, pour renvoyer l'utilisateur sur une page de remerciements ou de confirmation.

Dans le cas où le formulaire n'est pas validé, vous le renvoyez à l'utilisateur et en affichez les erreurs pour correction.

Enfin, dans le cas où la requête n'est pas de type `POST`, vous renvoyez le formulaire vide à l'utilisateur.

Affichage du formulaire

Vous venez de traiter les informations d'un formulaire rempli par l'utilisateur et de le renvoyer dans un template. Voyons maintenant comment l'afficher dans le template.

Cette partie reste assez simple. En effet, vous avez chargé l'objet `Form` dans le contexte. Il vous faut encore écrire dans le template le formulaire proprement dit.

```
<form action='' method='POST'>
{% csrf_token %}
{{form.as_ul}}
<input type='submit' value='valider' />
</form>
```

Dans ce court exemple, il faut remarquer plusieurs choses. La première directive crée un objet de type formulaire dans le navigateur. En définissant `action` à `''`, cela appelle l'URL sur laquelle se trouve votre internaute et servira à appeler la même fonction lors de la validation. La méthode est `POST` pour déclencher la partie de la vue qui valide et traite les données du formulaire – ces dernières ne sont traitées que si le formulaire est envoyé avec la méthode `POST`.

Le template tag `{% csrf_token %}` va générer un jeton d'identification unique pour protéger votre formulaire des attaques de type *Cross Site Request Forgery*. Dans Django, c'est une obligation pour toutes les requêtes de type `POST`.

Ensuite, vous appelez la méthode `as_ul` de votre formulaire : les champs de ce dernier seront affichés sous forme de liste. Django vous propose également les méthodes `as_p` et `as_table` pour présenter le formulaire respectivement sous forme de paragraphes et de tableau. Vous pouvez bien entendu définir votre propre type d'affichage.

Ce rapide tour d'horizon des formulaires sous Django vous montre la logique globale de leur fonctionnement. Nous allons maintenant voir comment prendre la main sur toutes les étapes, de la création de formulaires à leur affichage.

Les champs de formulaire

Pour modéliser un formulaire, le plus évident est de commencer par les champs d'entrée, qu'il s'agisse de champs textes, de listes de choix, de boutons radio ou de cases à cocher. Plusieurs aspects sont à prendre en compte : tout d'abord la façon dont ces formulaires sont affichés à l'utilisateur, puis dont les données entrées par l'utilisateur sont validées et, enfin, ce qu'il faut faire avec ces données (afficher une autre page, enregistrer en base de données…).

Il existe une corrélation assez étroite entre les champs des modèles et ceux des formulaires. Sur bien des points, ces deux concepts sont assez similaires. Comme pour les modèles, quand vous créez un formulaire, vous devez choisir le type des champs que vous souhaitez utiliser. Cela déterminera la manière dont le champ sera affiché à l'utilisateur, mais aussi validé.

Prenons un exemple simple : le champ de type `CharField`, qui est le pendant du champ `CharField` des modèles :

```
>>> from django import forms
>>> texte = forms.CharField()
```

Ce champ de formulaire très simple possède un widget responsable de l'affichage à l'utilisateur :

```
>>> texte.widget
<django.forms.widgets.TextInput at 0x108c6ded0>
```

Django a créé pour vous un widget de type `TextInput`. Il sera affiché à l'utilisateur sous la forme suivante :

```
<input type='text'>
```

Il possède également un validateur que vous pouvez invoquer avec sa méthode `clean` :

```
>>> text.clean('Bonjour')
u'Bonjour'
>>> text.clean('')
ValidationError: [u'This field is required.']
```

Le widget pour l'affichage et la méthode `clean` sont les deux éléments indispensables d'un champ de formulaire.

Les options remarquables d'un champ de formulaire

Lors de la création, vous pouvez facilement modifier la manière dont un champ de formulaire est validé ou affiché à l'utilisateur à l'aide d'options.

L'option required

Cette option modifie directement la manière dont le champ va être validé. Par défaut, elle prend la valeur `True`, c'est-à-dire que le champ est obligatoire : ne pas y insérer de valeur va engendrer une erreur de validation. Si vous souhaitez que l'utilisateur puisse ne pas renseigner ce champ, vous devez utiliser cette option de la façon suivante :

```
>>> texte = forms.CharField(required=False)
>>> texte.clean('')
u''
```

Le champ est vide mais, cette fois, il n'y a pas d'erreur de validation. Notez pourtant que si maintenant un champ vide est autorisé, cela ne désactive pas pour autant les autres validations du champ. Pour le constater, prenons un champ de type :

```
>>> email = forms.EmailField()
>>> email.clean('')
ValidationError: [u'This field is required.']
```

Il se comporte comme un `CharField`. Le widget utilisé pour l'afficher à l'utilisateur est d'ailleurs identique :

```
>>> email.widget
<django.forms.widgets.TextInput at 0x108c7e9d0>
```

La seule différence entre les types `EmailField` et `CharField` concerne la validation supplémentaire du format de l'e-mail :

```
>>> email.clean('un test')
[u'Enter a valid e-mail address.']
```

Vérifiez donc maintenant que l'option `required` n'annule pas la validation du format de l'e-mail :

```
>>> email = forms.EmailField(required='False')
>>> email.clean('')
u''
>>> email.clean('test')
ValidationError: [u'Enter a valid e-mail address.']
```

La seconde validation a donc bien été effectuée.

La validation des champs de formulaire

Vous pouvez donc avoir plusieurs validateurs pour un seul champ. Si la validation d'un champ vide est valable pour tous les types, le validateur de formatage d'e-mail est une spécificité du champ `EmailField`. Pour connaître les validateurs associés à un champ, utilisez l'attribut `validators` :

```
>>> email.validators
[<django.core.validators.EmailValidator at 0x10828e190>]
```

Lorsque vous instanciez un champ de formulaire, vous pouvez choisir les validateurs à employer. Ainsi, il est très simple de recréer de toutes pièces le fonctionnement d'un `EmailField` en partant d'un champ `CharField` :

```
>>> from django.core.validators import validate_email
>>> from django import forms
>>> email = forms.CharField(validators=[validate_email])
>>> email.clean('test')
ValidationError: [u'Enter a valid value.']
```

Écrire vos propres validateurs

Un validateur est une fonction qui prend en argument une valeur, c'est-à-dire ce que l'utilisateur a entré dans le champ, et renvoie une erreur de type `ValidationError` si le champ ne remplit pas certains critères.

Rapidement, vous souhaiterez sûrement écrire vos propres validateurs. Dans l'exemple qui va suivre, nous allons administrer un serveur d'e-mail et nous souhaitons que les utilisateurs puissent créer leurs adresses e-mails à l'aide d'un formulaire. Imaginons que vous possédez les noms de domaines `dupont.fr` et `dupond.fr`, vous voulez valider que l'adresse entrée par l'utilisateur :

- est bien sur l'un des deux noms de domaines ;
- n'est pas déjà utilisée ;
- correspond bien à une adresse valide.

Votre champ va donc utiliser trois validateurs. Il n'est pas utile d'écrire le dernier, car le champ `EmailField` se charge déjà de cette validation. Dans une utilisation réelle, les domaines que vous possédez et les noms d'utilisateurs seront stockés en base de données, mais pour simplifier les exemples, nous utiliserons de simples listes.

Le premier validateur vérifie l'appartenance aux deux noms de domaines :

```
>>> from django.core.exceptions import ValidationError
def validate_domain(value):
    domains = ['dupont.fr', 'dupond.fr']
    if len(value.split('@'))!= 2:
        pass
    elif not value.split('@')[1] in domains:
        raise ValidationError(u"Le domaine de l'adresse e-mail ne peut être que
dupont.fr ou dupond.fr")
```

Le second validateur s'assure que l'utilisateur n'existe pas déjà :

```
def validate_user(value):
    users = ['jean', 'pierre', 'marie']
    if len(value.split('@'))!= 2:
       pass
    if value.split('@')[0] in users:
        raise ValidationError(u"Le nom %s est déjà utilisé" %
value.split('@')[0])
```

Vous pouvez maintenant instancier votre nouveau champ :

```
>>> email = forms.EmailField(validators=[validate_domain, validate_user])
```

Vérifiez maintenant toutes les validations, d'abord l'adresse :

```
>>> email.clean('un simple test')
ValidationError: [u'Enter a valid e-mail address.']
```

Essayez maintenant de lever une erreur sur le nom de domaine :

```
>>> email.clean('test@gmail.com')
ValidationError: [u"Le domaine de l'adresse e-mail ne peut être que dupont.fr ou
dupond.fr"]
```

Testez ensuite la validation des utilisateurs :

```
>>> email.clean('jean@dupont.fr')
ValidationError: [u"Le nom jean est déjà utilisé"]
```

Enfin, validez que les erreurs peuvent être retournées en une seule fois :

```
>>> email.clean('jean@gmail.com')
[u"Le domaine de l'adresse e-mail ne peut être que dupont.fr ou dupond.fr", u"Le
nom jean est déjà utilisé"]
```

Les messages d'erreur

Lorsque vous créez des validateurs, vous pouvez choisir le message d'erreur que vous renvoyez à l'utilisateur. C'est ce que vous faites quand vous écrivez :

```
raise ValidationError(u"Le domaine de l'adresse e-mail ne peut être que
dupont.fr ou dupond.fr")
```

Pourtant, vous ne pouvez pas choisir de cette façon le message d'erreur des validateurs par défaut du champ que vous utilisez. Par exemple, le champ `EmailField` possède deux validateurs par défaut : `required` qui vérifie que le champ n'est pas vide et `invalid` qui vérifie la syntaxe de l'adresse. Pour modifier ces deux messages d'erreur, vous devez utiliser l'argument `error_messages`, qui est un dictionnaire ayant pour clés les noms des validateurs et pour valeurs les messages d'erreur que vous souhaitez voir s'afficher. Par exemple, votre champ de formulaire pourrait être écrit comme suit :

```
email = forms.EmailField(validators=[validate_domain, validate_user],
error_messages = {'required' : u"Vous devez préciser un e-mail.", 'invalid':
"Votre e-mail n'est pas correct."})
```

Les options d'affichage

Les options d'affichage contrôlent la manière dont l'information sera transmise à l'utilisateur. Elles sont au nombre de trois :

- `label`, qui contrôle le « titre » du champ tel qu'affiché à l'utilisateur ;
- `widget`, qui contrôle l'élément utilisé pour afficher les informations (texte, case à cocher, bouton radio…) ;
- `help_text`, qui affiche un texte descriptif du champ.

L'option `widget` est très riche et mérite une partie à part entière ; nous nous cantonnerons donc ici à `label` et `help_text`. Créez donc un formulaire très simple :

```
class FormExemple(forms.Form):
    nom = forms.CharField(label='Votre nom', help_text='Ce champ est
obligatoire. Il permet de vous identifier sur notre site Internet.')
```

Instanciez votre formulaire :

```
>>> exemple = FormExemple()
```

Affichez-le :

```
>>> exemple.as_ul()
u'<li>
  <label for="id_nom">Votre nom:</label>
  <input type="text" name="nom" id="id_nom" />
  <span class="helptext">
    Ce champ est obligatoire. Il permet de vous identifier sur notre site
Internet.
  </span>
</li>'
```

Les options `label` et `help_text` sont importantes pour vos utilisateurs. Elles simplifient la compréhension du formulaire que vous souhaitez les voir remplir. Ne les oubliez pas !

Les widgets

Django sélectionne pour vous un widget adapté suivant le type de champ. Ainsi, si vous utilisez un champ `BooleanField`, il définit un widget de type `CheckboxInput`, ce qui correspond à une case à cocher. Si la case est cochée, la valeur de retour sera `True` et, dans le cas contraire, elle sera `False`. Pour vous en assurer, vous pouvez tester ce champ de la façon suivante :

```
>>> test = forms.BooleanField()
>>> test.widget
<django.forms.widgets.CheckboxInput at 0x108caa0d0>
```

Vérifiez le rendu de ce champ en créant une classe formulaire et en utilisant sa méthode `as_ul` ou `as_p` ou encore `as_table` :

```
class TestForm(forms.Form):
    bool = forms.BooleanField()
>>>t = TestForm()
>>>t.as_p()
u'<p><label for="id_bool">Bool:</label> <input type="checkbox" name="bool"
id="id_bool" /></p>'
```

La plupart du temps, ce choix de widget est conforme à vos attentes. Parfois cependant, vous souhaiterez en utiliser un autre ou en modifier certaines caractéristiques

Sélectionner un widget particulier

Prenons l'exemple d'une liste de choix :

```
choix = (
    ('Mr', 'Monsieur'), ('Mlle', 'Mademoiselle'), ('Mme', 'Madame')
     )
```

Vous pouvez créer un champ de formulaire permettant de sélectionner l'un de ces choix de la manière suivante :

```
selection = forms.ChoiceField(choices=choix)
```

Django a sélectionné pour vous un widget de type `Select`.

```
>>> selection.widget
<django.forms.widgets.Select at 0x108cc2f10>
```

Il se présente à l'utilisateur de la façon suivante :

```
class Test(forms.Form):
     choix = (('Mr', 'Monsieur'), ('Mlle', 'Mademoiselle'), ('Mme', 'Madame'))
     selection = forms.ChoiceField(choices=choix)

>>> a = Test()
>>> a.as_p()
u'<p><label for="id_selection">Selection:</label> <select name="selection"
id="id_selection">\n<option value="Mr">Monsieur</option>\n<option
value="Mlle">Mademoiselle</option>\n<option value="Mme">Madame</option>\n</
select></p>'
```

L'utilisateur aura donc à sa disposition une liste déroulante. Si vous préférez proposer des boutons radio, il faut modifier l'argument `widget` de votre champ `selection` :

```
selection = forms.ChoiceField(
    choices=choix,
    widget=forms.RadioSelect)
```

En reprenant l'exemple du formulaire précédent, cela donne :

```
class Test(forms.Form):
     choix = (('Mr', 'Monsieur'), ('Mlle', 'Mademoiselle'), ('Mme', 'Madame'))
     selection = forms.ChoiceField(choices=choix, widget=forms.RadioSelect)

>>> a = Test()
>>> a.as_p()
u'<p><label for="id_selection_0">Selection:</label> <ul>\n<li><label
for="id_selection_0"><input type="radio" id="id_selection_0" value="Mr"
name="selection" /> Monsieur</label></li>\n<li><label
for="id_selection_1"><input type="radio" id="id_selection_1" value="Mlle"
name="selection" /> Mademoiselle</label></li>\n<li><label
for="id_selection_2"><input type="radio" id="id_selection_2" value="Mme"
name="selection" /> Madame</label></li>\n</ul></p>'
```

Modifier un widget existant

Parfois, votre besoin ne réside pas dans le choix du widget mais dans la manière dont il est affiché, dans les attributs qui le composent.

Tout d'abord, pour effectuer vos tests sur un widget, vous pouvez directement l'instancier ; vous n'êtes pas obligé de l'intégrer dans un champ de formulaire. De plus, les widgets bénéficient de la méthode `render` qui affiche directement leur forme HTML sans passer par une méthode de type `as_p` ou `as_ul` d'un formulaire complet. Les deux arguments obligatoires de la méthode `render` sont le nom du champ et sa valeur. Par exemple, pour tester un widget de type `TextInput`, vous pouvez écrire :

```
>>> from django import forms
>>> test = forms.TextInput()
>>> test.render('test', 'un test')
u'<input type="text" name="test" value="un test" />'
```

Pour modifier un widget existant, servez-vous de l'attribut `attrs`. Il s'agit d'un dictionnaire qui contient les différents arguments que vous souhaitez ajouter à la présentation HTML de votre widget. Par exemple, pour créer un widget de type `TextInput` de 250 caractères, vous pouvez écrire :

```
>>> test = forms.TextInput(attrs={'size':250})
>>> test.render("test", "un test")
u'<input type="text" name="test" value="un test" size="250" />'
```

De la même manière, vous pouvez préciser une classe particulière à utiliser pour afficher la représentation HTML de votre widget :

```
>>> test = forms.TextInput(attrs={'size':250, 'class': 'custom'})
>>> test.render("test", "un test")
u'<input value="un test" type="text" class="custom" name="test" size="250" />'
```

Une fois que votre widget vous convient, encapsulez-le dans un champ de formulaire, soit directement en ligne :

```
text = forms.CharField(
    widget=forms.TextInput(attrs={'class':'test'})
    )
```

soit, si vous utilisez souvent ce widget, en utilisant une variable :

```
text_widget = forms.TextInput(attrs={'class':'test'})
text = forms.CharField(widget=text_widget)
```

soit en utilisant le dictionnaire de la classe `Meta` du formulaire :

```
class MyForm(forms.Form):
    text = forms.CharField()
    class Meta:
    widgets = {
            'text': forms.TextInput(attrs={'class':'test'}))
        }
```

Créer un widget personnalisé

La palette de possibilités que vous venez de voir vous offre un grand contrôle sur vos champs de formulaires. Pourtant, il est possible que vous souhaitiez créer vos propres widgets, parce que, par exemple, vous souhaitez définir une logique de validation particulière, utiliser une feuille de style spéciale sur ce formulaire ou une fonction JavaScript spécifique. Vous pouvez aussi choisir de créer un widget spécifique pour ne pas avoir besoin de redéfinir des options particulières sur de nombreux champs. Quelles qu'en soient les raisons, là encore Django vous offre une manière simple et élégante pour le faire.

Tout d'abord, nous ne parlerons que de la création d'un widget de toutes pièces. Vous n'utiliserez pas de widget existant, mais uniquement la classe `Widget`, qui est une classe de base où toute la logique est à votre charge. Elle possède l'attribut `attrs` que vous avez déjà manipulé, mais ne possède pas de méthode `render`. C'est donc à vous de la créer.

La méthode `render` prend en arguments :

- le nom du champ ;
- la valeur du champ ;
- un dictionnaire optionnel d'arguments : `attrs`.

La méthode `render` doit renvoyer un caractère Unicode correspondant à la représentation HTML de votre widget.

Pour vous aider dans votre tâche consistant à implémenter `render`, la classe `Widget` vous fournit également la méthode `build_attrs` pour former un dictionnaire d'attributs utiles pour la création de votre champ de formulaire HTML. De plus, comme vous ne maîtrisez pas du tout ce que l'utilisateur va bien pouvoir entrer dans le formulaire, vous devez mettre en place un système assez robuste pour parer à toute éventualité. Par exemple, Django définit le widget `TextInput` à peu près comme suit :

```
class TextInput(Widget):

    input_type = 'text'

    def render(self, name, value, attrs=None):
```

```
        if value is None:
            value = ''
        final_attrs = self.build_attrs(
                      attrs,
                      type=self.input_type, name=name)
        if value != '':
            final_attrs['value'] = force_unicode(value)
        return mark_safe(u'<input%s />' % flatatt(final_attrs))
```

La méthode `flatatt` est issue de `django.forms.util` et sert à transformer un dictionnaire, par exemple :

```
{'name' : 'test', value : 'Une valeur'}
```

en sa contrepartie HTML pour un champ de formulaire, soit :

```
name='test' value='Une valeur'
```

La méthode `render` est donc la seule qui soit vraiment obligatoire pour que votre widget fonctionne. Pourtant, si vous en êtes à devoir créer votre propre outil, vous souhaiterez probablement surclasser les widgets existants. Pour ce faire, il faut observer la manière dont chacun est écrit (https://github.com/django/django/blob/master/django/forms/widgets.py), le surclasser et modifier la méthode qui vous intéresse.

Modifier l'affichage HTML d'un formulaire

Lorsque vous utilisez les fonctionnalités de `form.as_p` ou de `form.as_ul`, par exemple, vous laissez Django se charger de l'affichage du formulaire. Bien entendu, il vous est tout à fait possible de prendre intégralement le contrôle de ce dernier.

Par exemple, définissez un formulaire de contact classique comme suit :

```
class Contact(forms.Form) :
    nom = forms.CharField()
    prenom = forms.CharField()
    message = forms.CharField()
    email = forms.EmailField()
```

Avec la méthode `as_ul`, vous obtenez le rendu HTML suivant :

```
<li>
  <label for="id_nom">Nom:</label>
  <input type="text" name="nom" id="id_nom" />
</li>
```

```
<li>
  <label for="id_prenom">Prenom:</label>
  <input type="text" name="prenom" id="id_prenom" />
</li>
<li>
  <label for="id_message">Message:</label>
  <input type="text" name="message" id="id_message" />
</li>
<li>
  <label for="id_email">Email:</label>
  <input type="text" name="email" id="id_email" />
</li>
```

Pour rendre vous-même ce formulaire, vous allez utiliser les variables que vous offre Django. En imaginant que votre contexte contient `{ 'form' : Contact }`, vous avez à votre disposition :

- `{{ form.non_field_errors }}` qui contient toutes les erreurs qui ne sont pas liées à un champ de formulaire particulier (par exemple, l'erreur que vous rencontrez lorsqu'un champ de confirmation ne correspond pas au champ dont il dépend) ;
- `{{ form.<champs>.errors }}` qui correspond aux erreurs de validation du champ en question (dans notre exemple, vous aurez donc `form.nom.errors`, `form.prenom.errors`, `form.message.errors` et `form.email.error`) ;
- `{{ form.<champs>.label }}` qui correspond au label de votre champ (par exemple, `message :` pour le champ `message`) ;
- `{{ form.<champs>.label_tag }}` qui vous renvoie le code HTML du label (par exemple, `<label for="id_nom">Nom:</label>` pour le champ `nom`) ;
- `{{ form.<champ> }}` qui correspond au champ de formulaire de votre champ.

Vous pouvez donc écrire :

```
<form action ='' method='POST' >
  {{ form.non_field_errors }}
  <ul>
    <li>
      {{ form.nom.errors }}
      {{ form.nom.label_tag}}
      {{ form.nom}}
    </li>
    <li>
      {{ form.prenom.errors }}
      {{ form.prenom.label_tag}}
      {{ form.prenom}}
    </li>
    <li>
      {{ form.message.errors }}
```

```
      {{ form.message.label_tag}}
      {{ form.message}}
    </li>
    <li>
      {{ form.email.errors }}
      {{ form.email.label_tag}}
      {{ form.email}}
    </li>
</form>
```

Dans cette écriture, nous ne modifions rien. L'idée est de vous montrer comment vous pouvez tirer parti des variables que vous offre Django pour, par exemple, ajouter des tags HTML spécifiques sur certains champs ou regrouper les erreurs dans un bloc particulier… Dans le cas où les modifications que vous souhaitez faire sont identiques quel que soit le champ, il est dommage de devoir, comme ici, copier-coller des blocs quasiment identiques. Il vous est heureusement possible de bénéficier du même type d'affichage mais, cette fois, en utilisant une itération sur l'ensemble des champs. Avec un formulaire nommé `form`, vous pouvez donc écrire :

```
{{ form.non_field_errors }}
<ul>
{ % for field in form %}
<li>
{{ field.errors }}
{{ field.label_tag }}
{{ field}}
</li>
{ % endfor %}
</ul>
```

Cette écriture, plus concise, aura le même effet que le code précédent. C'est donc à vous de décider de la forme à adopter en fonction de vos besoins en termes d'affichage de formulaire.

Conclusion

La gestion des formulaires est un point crucial dans la conception d'une application web. Gardez toujours à l'esprit que le dialogue avec l'utilisateur va définir la relation que vous entretiendrez avec lui. Vous devez donc soigner tout particulièrement l'affichage des formulaires et des éventuelles erreurs. N'oubliez pas également que vous ne devez jamais faire confiance aux données de l'utilisateur, et ce, pour des raisons de sécurité évidentes.

14

Django et le protocole web

Django est un framework web. Vous aurez donc à manipuler des concepts qui sont liés au Web et à ses protocoles. Dans ce chapitre, nous vous proposons donc de revenir sur les fondamentaux du Web et de détailler comment Django implémente ces concepts.

Les verbes du Web

Dans les précédents chapitres, nous avons évoqué à plusieurs reprises `GET` et `POST`, qui sont les verbes les plus couramment employés par le protocole HTTP. Par ailleurs, il est de plus en plus fréquemment fait usage des autres verbes HTTP que sont `PUT`, `DELETE` et `PATCH`. Ces cinq verbes forment un ensemble fortement utilisé dans un contexte REST.

GET et POST

HTTP (*HyperText Transfer Protocol*) est une norme, une manière de traduire le Web. Pour HTTP, les pages Internet sont des ressources. Elles sont caractérisées par un identifiant qui se doit d'être unique ; on parle d'URI (*Unique Resource Identifier,* identifiant unique de ressource). Si ce terme ne vous est pas familier, vous connaissez certainement mieux sa « petite sœur » URL (*Unique Resource Locator,* localisateur unique de ressource). Que l'on parle d'URI ou d'URL, au niveau d'HTTP cela ne fait pas de différence. Vous accédez à une ressource (une page web) par son URL, qui se doit d'être unique. Toutefois, accédez-vous à cette ressource pour la lire ou pour la

modifier ? C'est pour faire cette distinction qu'existent les verbes GET et POST : le premier permet de lire une ressource, le second de la modifier. Ils ont l'avantage d'être compris directement par les navigateurs. Ainsi une requête du type :

```
GET http://localhost:8000/articles/?paru=true&title=django
```

peut se traduire par : « récupérer pour lecture http://localhost:8000/articles/ avec les arguments paru=true et titre=django ».

Votre application devrait être en mesure de retrouver les articles dont le titre est « Django » et qui ont le statut « paru ». En revanche :

```
POST http://localhost:8000/articles/ arguments : paru=true et titre=django
```

est d'un fonctionnement très différent, puisque l'on cherche à modifier la ressource articles en lui donnant l'état paru et le titre django.

Pour votre usage, en tant que développeur web, vous devez prendre garde à respecter cette convention. Pour savoir quand utiliser GET ou POST, demandez-vous simplement si vous modifiez la ressource. Ainsi, un formulaire de recherche, aussi complexe soit-il, devra être appelé avec la méthode GET. En revanche, un formulaire qui permet de modifier un contenu devra toujours utiliser la méthode POST.

Les raisons de ces choix sont multiples. Si vous faites une requête GET, les paramètres de la requête sont compris dans l'URL. Il est donc simple pour un utilisateur d'enregistrer une recherche dans ses favoris ou de la partager en copiant-collant l'URL du navigateur. En revanche, dans le cas d'une requête POST, les arguments de la requête ne sont pas dans l'URL ; cela évite les erreurs de manipulation. De plus, en utilisant Django, vous bénéficiez également de la protection CSRF (*Cross Site Request Forgery* : attaque consistant à « voler » la session d'un utilisateur pour valider à sa place un formulaire) sur les requêtes de type POST. Enfin, vous aurez un code plus lisible et vous pourrez déterminer du premier coup d'œil l'usage d'une fonction simplement en observant le verbe utilisé.

Les verbes REST

REST signifie *Representational State Transfert*. Il se base sur des requêtes sans état *(stateless)*, c'est-à-dire qu'il n'existe aucun lien entre les requêtes. Si vous souhaitez créer, lire, modifier ou supprimer une ressource, vous n'avez qu'une seule action à réaliser. REST n'est ni un protocole, ni un format : c'est un type d'architecture. Vous l'utiliserez dans le domaine du Web, mais vous pourriez tout aussi bien l'employer dans d'autres domaines. On peut aisément représenter une requête de la manière suivante : VERBE URL arguments.

Si GET et POST sont utiles pour savoir ce que doit faire la requête, c'est un peu limité pour créer une architecture complète. C'est pourquoi d'autres verbes peuvent venir à notre secours.

- PUT indique qu'il faut créer une nouvelle ressource.
- DELETE précise qu'il faut supprimer une ressource.
- HEAD s'utilise comme GET, à la différence qu'il ne renvoie pas le contenu de la ressource, mais seulement les en-têtes. Il sert, par exemple, pour connaître la date de la dernière modification de la ressource, si elle existe (grâce au *status code* présent dans les en-têtes).
- PATCH modifie un ensemble de ressources.

Contrairement à GET et POST, ces verbes ne sont pas reconnus par les navigateurs. Ainsi, vous les trouverez principalement utilisés dans les API REST, où leur usage s'est fortement répandu.

Un exemple concret : mise en place d'une API complète

Comme pour toute architecture, il est souvent plus simple de la pratiquer que d'en parler, pour en comprendre toutes les subtilités. Aussi allons-nous mettre en place pour l'exemple une API complète : client et serveur. Nous allons donc mener deux projets distincts. Le premier sera le producteur de ressource et le second un consommateur de ladite ressource.

Pour accéder à l'API, vous allez utiliser la méthode d'authentification OAuth2. C'est un protocole très puissant et souple basé sur le concept de publication et le partage de ressources sécurisées. Beaucoup plus simple à mettre en place que OAuth1, OAuth2 offre une sécurité équivalente à sa première version.

Afin de corser un peu l'exercice, nous implémenterons cette API au-dessus du CMS Mezzanine (http://mezzanine.jupo.org), très performant et facilement modifiable. Il s'agira donc dans les prochaines sections de mettre en place une API pour le module de blog de Mezzanine.

Installation de Mezzanine

Tout d'abord, comme pour tout nouveau projet, il faut créer puis activer un environnement virtuel :

```
virtualenv api_tutorial
```

Sous Linux et Mac OS X :

```
source api_tutorial/bin/activate
```

Sous Windows :

```
api_tutorial\Scripts\activate.bat
```

L'installation de Mezzanine est assez simple :

```
pip install mezzanine
mezzanine-project testapi
cd testapi
python manage.py createdb
python manage.py runserver
```

Ouvrez ensuite votre navigateur à l'adresse `http://localhost:8000` comme vous en avez l'habitude et explorez les possibilités de ce CMS.

Description de OAuth2

L'objectif maintenant est de mettre en place une API pour récupérer, modifier, créer et supprimer les articles de blog de Mezzanine. Le concept d'OAuth est de gérer les autorisations à trois tiers.

- Le premier c'est Mezzanine, qui est le *provider*, c'est-à-dire celui qui produit la ressource.
- Le deuxième c'est l'utilisateur à qui appartient la ressource.
- Le dernier c'est le service externe qui souhaite manipuler les ressources de l'utilisateur.

Afin de clarifier les termes employés, on parle donc de :

- **provider** : Mezzanine ;
- **client** : l'application tierce ;
- **utilisateur** : l'utilisateur de Mezzanine qui veut laisser le client manipuler ses articles.

Création de l'authentification OAuth2

Pour mettre en place la sécurité de votre application, vous aurez besoin d'un module qui va gérer les autorisations que vos utilisateurs vont donner à des clients tiers. Pour cela, il faut installer le module `django-oauth2-provider` :

```
pip install django-oauth2-provider
```

Lorsque c'est fait, il faut encore l'ajouter à vos `INSTALLED_APPS` dans `settings.py` et ajouter les URL de cette application dans le fichier `urls.py` à la racine de votre projet :

```
url(r'^oauth2/', include('provider.oauth2.urls',namespace='oauth2')),
```

Afin de créer les tables dans votre base de données, exécutez :

```
python manage.py syncdb
```

Si vous vous connectez ensuite à votre interface d'administration, vous verrez une nouvelle entrée de menu : *Oauth2*, contenant elle-même l'entrée *Client* permettant de créer des comptes clients d'accès aux ressources. Nous y reviendrons dès qu'il s'agira d'implémenter le client.

Création de l'API

Aujourd'hui, l'écosystème Django offre des modules très performants pour créer des API REST. Nous avons choisi de vous faire découvrir `tastypie`, qui est à la fois robuste, souple et simple d'utilisation.

Commencez par l'installer :

```
pip install django-tastypie
```

Puis ajoutez `tastypie` à vos `INSTALLED_APPS`.

Pour fonctionner, `tastypie` a besoin d'un fichier API au sein de l'application qui va permettre de configurer les différents aspects de l'API. Comme l'application `blog` de Mezzanine est externe et que votre API est un projet séparé, nous vous proposons de créer une nouvelle application dont l'objectif sera d'étendre `blog` de Mezzanine pour lui ajouter une API REST.

```
python manage.py startapp blog_api
```

Pensez à ajouter cette application à vos `INSTALLED_APPS`.

Maintenant que tout est prêt, vous pouvez commencer à décrire l'API du blog :

```
from tastypie.resources import ModelResource
from mezzanine.blog.models import BlogPost
```

Très similaire à celle des modèles de Django, la syntaxe de `tastypie` sert à créer des modèles de « ressources » qui seront rendus disponibles au travers d'une API. Dans une version très simple, la description d'un article de blog peut se faire de la manière suivante :

```
class PostRessource(ModelResource):
    class Meta:
        queryset = BlogPost.objects.all()
```

Pour rendre disponible ce modèle dans une API, il faut lui donner une URL. Dans le fichier racine de votre projet, vous pouvez donc ajouter :

```
("^api/", include("blog_api.urls")),
```

Ensuite, dans le fichier `blog_api/urls.py`, vous pouvez écrire :

```
from django.conf.urls.defaults import patterns, include, url
from tastypie.api import Api
from api import PostRessource

v1_api = Api(api_name='v1')
v1_api.register(PostRessource())

urlpatterns = patterns("",
        url(r'^', include(v1_api.urls)),
)
```

Votre API est maintenant disponible sous l'URL `/api/v1/` et elle peut afficher les ressources demandées soit en XML soit en JSON, pour lire l'ensemble des articles du blog :

```
http://localhost:8000/api/v1/postressource/?format=json
```

Sécuriser l'API

Si ce premier exemple très rapide est fonctionnel, il n'est protégé par aucune sécurité. Tout au plus `tastypie` interdit-il par défaut la modification ou la suppression des articles au travers de l'API. La lecture, elle, est autorisée pour tous.

Utilisons OAuth2 pour sécuriser cette API. Au même niveau que votre fichier `api.py`, écrivez un fichier `auth.py` dont le contenu sera le suivant :

```
from tastypie.authentication import Authentication
from tastypie.authorization import Authorization
from provider.oauth2.models import AccessToken
import datetime
from django.utils.timezone import utc
```

L'authentification de `tastypie` fait usage de deux classes : `Authentication`, qui indique si une requête est authentifiée ou non, et `Authorization`, qui précise si la personne authentifiée ou anonyme a les droits suffisants pour manipuler la ressource demandée. Ici, nous souhaitons faire en sorte que le module `oauth2` précédemment installé et configuré se charge de cette vérification.

Authentification

Pour être authentifiée par OAuth2, une requête doit être munie d'un paramètre `access_token`, qui sert à identifier le client et l'utilisateur qui lui a fourni l'autorisation. Ce paramètre précise également le niveau de droit que l'utilisateur a fourni à l'application tierce : lecture, écriture ou lecture et écriture.

La méthode indiquant si la requête est authentifiée est `is_authenticated`, qui se présente sous la forme suivante :

```
class Oauth2Authentication(Authentication):
    def is_authenticated(self, request, **kwargs):
        if "access_token" in request.GET:
            if AccessToken.objects.get(
                token=request.GET['access_token']
                ):
                return True

        return False
```

Ainsi, si l'on parvient à trouver le *token* de la requête dans la liste des tokens connus, on considère que la requête est bien authentifiée et on renvoie `True`. Dans tout autre cas, on renvoie `False`.

Il vous sera utile également de savoir qui est authentifié. Pour cela, il faut implémenter la méthode `get_identifier`, qui prend la syntaxe suivante :

```
    def get_identifier(self, request):

        if "access_token" in request.GET:
            access_token = AccessToken.objects.get(
                token=request.GET['access_token']
                )

            return access_token.user
```

Maintenant que l'on sait si l'utilisateur est authentifié ou non, il reste à s'assurer qu'il a les droits suffisants pour accéder à la ressource. Pour cela, c'est la classe `Authorization` qui entre en jeu :

```
class Oauth2Authorization(Authorization):

    def is_authorized(self, request, object=None):
        if "access_token" in request.GET:
            access_token = AccessToken.objects.get(
                token=request.GET['access_token']
                )

       if access_token.expires > datetime.datetime.utcnow().replace(tzinfo=utc):
            return True
        else:
            return False
```

Si vous utilisez Django 1.3, écrivez :

```
if access_token.expires > datetime.datetime.now()
```

au lieu de :

```
if access_token.expires > datetime.datetime.utcnow().replace(tzinfo=utc):
```

Limiter l'accès aux ressources

La dernière chose qui vous reste à faire est de limiter l'accès aux ressources, c'est-à-dire de faire en sorte qu'un utilisateur ne puisse manipuler que ses propres articles :

```
    def apply_limits(self, request, object_list):

        if "access_token" in request.GET:
            access_token = AccessToken.objects.get(
                token=request.GET['access_token']
                )
            return object_list.filter(
                user__username=access_token.user.username)

        return object_list.none()
```

Créer un client

Maintenant que votre API est créée, encore faut-il pouvoir l'utiliser. Pour ce faire, vous devez, dans l'interface d'administration, créer un client en suivant les liens : *Oauth2>Client>Add Client* et en remplissant les champs.

Pour les champs *url du site* et *url de redirection*, comme vous utilisez cette application en local et pour vous-même, indiquez une adresse du type `http://www.exemple.com`. Une fois votre application cliente réellement en production, vous pourrez changer ces

champs pour une véritable adresse. Puisqu'il s'agit d'une application de test, vous pouvez choisir indifféremment le type *confidentiel* ou *public*. Une fois le client sauvegardé, notez quelque part votre `client id` et votre `client secret`.

Maintenant que vous en êtes à implémenter un client qui va accéder à l'API, vous devez changer de répertoire (ouvrir un nouveau terminal, par exemple) et créer un nouvel environnement virtuel :

```
virtualenv blog_client
```

La première chose que votre application cliente va devoir faire, c'est demander à votre utilisateur l'autorisation d'accéder à ses ressources présentes sur l'application Mezzanine. Pour ce faire, vous allez utiliser le module Python `requests-oauth2`, qui facilite le dialogue d'un client vers un serveur OAuth2.

```
pip install requests-oauth2
```

Pour les premiers tests, vous pouvez ouvrir un interpréteur Python dans votre terminal :

```
>>> from requests_oauth2 import Oauth2
```

Vous devez ensuite instancier un client `oauth` qui prendra en argument votre `client id`, votre `client secret`, l'URL de l'API et les URL de l'API qui gèrent l'autorisation et les `access_token`. L'application `provider.oauth2` implémente les deux URL suivantes :

- `oauth2/authorize`
- `oauth2/access_token`

Vous pouvez donc instancier votre objet `Oauth2` comme suit :

```
>>> client_oauth = Oauth2(id_client, client_secret, 'localhost:8000/', 'oauth2/
authorize/', 'oauth2/access_token')
```

Utilisez ensuite la méthode `authorize_url` de OAuth2 qui va créer une URL que l'utilisateur devra entrer dans son navigateur. Dans une application web classique, vous le redirigerez tout simplement sur cette URL. La méthode `authorization_url` doit contenir les paramètres `response_type` et `scope`. Le premier correspond à ce que l'on attend du serveur. Ce peut être d'obtenir un code pour demander un token ou pour rafraîchir un token arrivé à expiration. Le second définit le niveau de droits que l'on demande (lecture, écriture ou lecture et écriture).

```
>>> authorization_url = client_oauth.authorize_url(response_type ='code',
scope='write'
```

Vous pouvez maintenant afficher `authorization_url` et vous rendre à cette URL avec votre navigateur. L'application Mezzanine va alors vous afficher un formulaire qui vous demande si vous acceptez ou non que l'application cliente accède à vos ressources. Si vous l'autorisez, vous serez redirigé vers l'URL de redirection indiquée.

Sur la page où vous vous trouvez actuellement, vous voyez dans la barre d'adresse une URL de ce type : `http://www.exemple.com/?state=&code=4b913ff0e982d51efb807a79c2ffa3cf650130df`. Notez bien le paramètre `code`, car vous en aurez besoin pour contacter le serveur et lui demander le précieux token.

De retour dans le terminal, vous pouvez maintenant utiliser la méthode `get_token` prenant en argument le fameux code et le `grant_type`, qui correspond à ce que vous souhaitez recevoir du serveur. Ici, il s'agit d'obtenir un jeton d'autorisation aussi appelé `authorization_code` :

```
>>> response = oauth2_handler.get_token(code, grant_type="authorization_code")
```

La réponse devrait ressembler à ce qui suit :

```
{u'access_token': u'8bc1ba0197e53093dfda807ed0dbe48e2f6571a5',
u'scope': u'read',
u'expires_in': 86399,
u'refresh_token': u'b43d606a8e50406ad1c99ff9b50c23d29eb3c96f'}
```

Gardez précieusement la clé `access_token` car c'est elle qu'il faudra employer pour chacune des requêtes sur l'API. Voici succinctement comment, avec la bibliothèque `requests`, vous utiliserez ce token :

```
>>> import requests
>>> oauth2_client = requests.session(params={'access_token' : access_token}
```

Votre client `oauth` est maintenant en mesure de faire des requêtes. Par exemple :

```
>>> oauth2_client.get(http://localhost:8000/api/v1/?format=json)
```

ou

```
>>> oauth2_client.get(http://localhost:8000/api/v1/postressource/?format=json)
```

slumber un client d'API REST

Maintenant que vous avez à votre disposition votre token d'authentification, vous pourriez sans difficultés effectuer vos requêtes simplement à l'aide de votre nouveau client `oauth`. Pourtant, afin de simplifier l'écriture du code, nous allons utiliser un

client en Python basé sur le module `requests` et qui est spécialement conçu pour cette tâche : `slumber`.

C'est un projet assez récent mais qui, jour après jour, gagne en utilisateurs. Il s'interface particulièrement bien avec `tastypie`, même si cette bibliothèque n'est pas requise pour utiliser `slumber`. Ce que vous venez d'apprendre est d'ailleurs tout à fait compatible avec les services OAuth2 tels que les API Google ou Facebook. Néanmoins, `slumber` ne s'occupe pas d'authentification, du moins pas d'authentification OAuth2. Pourtant, comme cette application repose sur `requests`, le module que vous utilisez depuis le début de ce tutoriel, il est très simple de gérer ce cas en quelques lignes :

```
>>> session = requests.session(params={'access_token' : access_token}
```

Vous souvenez-vous de cette ligne ? C'est tout simplement la même que celle qui permet de manipuler l'API Mezzanine. Nous gardons de côté cette variable `session` pour l'utiliser plus loin. Il est temps maintenant de configurer `slumber` :

```
>>> import slumber
>>> api = slumber.API('http://localhost:8000/api/v1/')
```

Vous créez une instance de `slumber.API` qui prend pour unique argument l'URL de votre API.

```
>>> api._store['session'] = session
```

C'est ici que se cache l'astuce : vous chargez la session que vous avez créée directement dans le dictionnaire `store` de `slumber.api`. Désormais, vous pouvez employer cette API de la façon la plus simple qui soit. Par exemple, pour récupérer la liste de tous vos posts, écrivez :

```
>>> api.postressource.get()
```

En bref

Voyez comment, en quelques manipulations, vous avez été en mesure de :

- mettre en place un système d'authentification OAuth2 ;
- mettre en place une API REST ;
- créer un client d'authentification OAuth2 ;
- créer un client d'API REST utilisant les requêtes OAuth2.

Dans la prochaine section, nous irons encore plus loin avec notre API, en installant les autres verbes, car il est vrai que, pour le moment, cette dernière n'est en mesure

d'effectuer que des requêtes de type `GET`. Il vous faut encore apprendre à filtrer les résultats, récupérer un enregistrement unique, et effectuer des modifications, créations et suppressions sur vos données. Rassurez-vous toutefois, vous venez de faire le plus dur ! Et ce tutoriel vous sera d'une aide précieuse lorsque vous souhaiterez mettre en place une API sécurisée pour votre propre application ou tout simplement créer un client pour les API mondialement connues comme Facebook ou Google.

Une API REST en détail

Maintenant que votre API REST est opérationnelle, il vous reste à l'enrichir et à tester son fonctionnement au travers de votre client. `tastypie` et `slumber` vous seront d'une aide précieuse.

La première chose à faire est de récupérer un objet unique. La bonne nouvelle ici est que cette possibilité vous est déjà offerte. Malheureusement, il faut bien reconnaître qu'il est difficile de lire les résultats JSON non formatés de la commande suivante :

```
>>> api.postressource.get()
```

Nous allons donc, dans un premier temps, clarifier un peu l'affichage avec une petite bibliothèque fort pratique : `pprint`, qui fait partie de la bibliothèque standard – vous pouvez donc l'utiliser sans craintes pour tous vos projets. En voici la syntaxe :

```
>>> import pprint
>>> pp = pprint.PrettyPrinter(indent=2)
>>> results = api.postressource.get()
>>> pp.pprint(results)
```

Comme vous le constatez, vous pouvez choisir le niveau d'indentation.

Retrouver une ressource unique

Vous pouvez maintenant facilement distinguer l'architecture de la réponse. Elle contient tout d'abord une clé `meta` qui vous informe sur la requête elle-même. Par exemple, comme nous n'avons pas modifié le comportement par défaut de `tastypie`, la réponse contient tout au plus 20 résultats et vous offre la possibilité, si votre blog contient plus de 20 articles, d'obtenir les résultats suivants grâce à une pagination. Vous remarquerez également que chaque ressource, ici des articles de blog mais il en serait de même pour n'importe quel enregistrement, possède une clé `resource_uri`. Il s'agit là de l'identifiant unique désignant sans ambiguïté la ressource que l'on traite. Elle est ici de la forme suivante :

```
/api/v1/postressource/5/
```

Comme `slumber` veut vous simplifier la vie, vous pouvez simplement écrire :

```
api.postressource(5).get()
```

Cela se traduira par la requête HTTP suivante :

```
GET http://localhost:8000 /api/v1/postressource/5/
?access_token=8bc1ba0197e53093dfda807ed0dbe48e2f6571a5
```

Bien entendu, votre token sera différent. L'important est que ce soit bien l'identifiant unique qui sera employé pour manipuler la ressource.

Créer une nouvelle ressource

Maintenant que vous êtes en mesure d'obtenir une ressource unique, vous voudrez également en créer d'autres (nouveaux articles de blog). Avant cela, il vous faudra modifier un peu votre API. En effet, si vous avez mis en place une authentification OAuth2 pour ne renvoyer au client que les articles de blog qu'il a lui-même écrits, il faut également que ce filtre soit effectif à la création d'une nouvelle ressource. En d'autres termes, il faut que le nouvel article créé par l'utilisateur appartienne à ce même utilisateur. Pour cela, vous allez utiliser la méthode `hydrate` de la classe `ModelResource` fournie par `tastypie`.

Cette méthode sert à effectuer des actions avant que l'objet ne soit enregistré en base de données. Elle prend en argument `bundle`, qui est un objet contenant, entre autres, la requête de l'utilisateur. Vous pourrez ainsi retrouver l'utilisateur qui a effectué cette requête grâce à l'`access_token`.

Donc, dans votre fichier `api.py`, commencez par importer la classe `AccessToken` :

```
from provider.oauth2.models import AccessToken
```

Votre classe `PostRessource` peut maintenant implémenter une méthode `hydrate` :

```
def hydrate(self, bundle):
    if "access_token" in bundle.request.GET:
        access_token = AccessToken.objects.get(
            token=bundle.request.GET['access_token']
                )
        bundle.obj.user = access_token.user
        return bundle
```

Dans cette méthode, nous utilisons le token (ou jeton) pour retrouver l'utilisateur auquel il appartient. Comme `bundle` contient l'objet en cours de création, vous pouvez lui affecter l'utilisateur que vous venez de trouver, et le tour est joué !

Il reste maintenant à tester votre code au niveau de votre client. Comme nous sommes dans le contexte d'une API REST, c'est le verbe utilisé qui indique à l'API ce qu'il faut faire. Ici, nous souhaitons créer une nouvelle ressource ; il faut donc employer le verbe `POST` et fournir tous les champs obligatoires. Pour un article de blog sur le CMS Mezzanine, vous avez besoin des champs `title` et `content`. Le champ `auteur` est lui aussi obligatoire, mais nous venons de créer une fonction qui l'ajoutera elle-même ; ce n'est donc plus la peine de s'en préoccuper. Toujours à l'aide du client `Slumber`, écrivez :

```
>>> nouvel_article = api.postressource.post({'title' : 'mon nouvel article',
'content' : 'ici le contenu de mon article'})
```

Pour vérifier que la création du nouvel article s'est bien déroulée, affichez la variable `nouvel_article` :

```
>>> pp.pprint(new)
{ u'_meta_title': None,
  u'allow_comments': True,
  u'comments_count': 0,
  u'content': u'ici le contenu de mon article',
  u'description': u'ici le contenu de mon article',
  u'expiry_date': None,
  u'featured_image': None,
  u'gen_description': True,
  u'id': u'10',
  u'keywords_string': '',
  u'publish_date': u'2012-10-09T07:56:25.779764+00:00',
  u'rating_average': 0.0,
  u'rating_count': 0,
  u'resource_uri': u'/api/v1/postressource/10/',
  u'short_url': None,
  u'slug': u'mon-nouvel-article',
  u'status': 2,
  u'title': u'mon nouvel article'}
```

Modifier une ressource

Maintenant que vous êtes en mesure de récupérer des objets et que vous pouvez également en créer, il vous faut aussi pouvoir les modifier. Pour ce faire, le protocole HTTP a prévu `PUT`. Il sert à mettre à jour une ressource, c'est-à-dire qu'il va remplacer les attributs de la ressource par ceux du dictionnaire que vous lui soumettez.

Testons cette fonction. Tout d'abord, récupérons l'article que vous venez de créer ; sa `resource_uri` est dans notre exemple `/api/v1/postressource/10/`, mais sera sans doute différent dans votre version :

```
>>> api.postressource(10).get()
{u'status': 2, u'content': u'ici le contenu de mon article', u'allow_comments':
True, u'description': u'ici le contenu de mon article', u'rating_count': 0,
u'rating_average': 0.0, u'slug': u'mon-nouvel-article', u'title': u'mon nouvel
article', u'keywords_string': '', u'_meta_title': None, u'expiry_date': None,
u'comments_count': 0, u'featured_image': None, u'short_url': None,
u'publish_date': u'2012-10-09T07:56:25.779764+00:00', u'id': u'10',
u'gen_description': True, u'resource_uri': u'/api/v1/postressource/10/'}
```

Maintenant, renvoyez l'ensemble de la réponse à l'aide de la méthode `PUT`, mais en modifiant le champ `title` :

```
>>> api.postressource(10).put({u'status': 2, u'content': u'ici le contenu de mon
article', u'allow_comments': True, u'description': u'ici le contenu de mon
article', u'rating_count': 0, u'rating_average': 0.0, u'slug': u'mon-nouvel-
article', u'title': u'A propos de Slumber', u'keywords_string': '',
u'_meta_title': None, u'expiry_date': None, u'comments_count': 0,
u'featured_image': None, u'short_url': None, u'publish_date': u'2012-10-
09T07:56:25.779764+00:00', u'id': u'10', u'gen_description': True,
u'resource_uri': u'/api/v1/postressource/10/'})
True
```

Comme vous pouvez le voir, le serveur vous répond `True`, ce qui laisse supposer que la modification du titre a bien été effectuée. Pour en avoir le cœur net, refaites un `GET` sur votre ressource :

```
>>>api.postressource(10).get()
{u'status': 2, u'content': u'ici le contenu de mon article', u'allow_comments':
True, u'description': u'ici le contenu de mon article', u'rating_count': 0,
u'rating_average': 0.0, u'slug': u'mon-nouvel-article', u'title': u'A propos de
Slumber', u'keywords_string': '', u'_meta_title': None, u'expiry_date': None,
u'comments_count': 0, u'featured_image': None, u'short_url': None,
u'publish_date': u'2012-10-09T07:56:25.779764+00:00', u'id': u'10',
u'gen_description': True, u'resource_uri': u'/api/v1/postressource/10/'}
```

La ressource a bien été modifiée. Il est aussi possible de ne renvoyer que le titre :

```
>>> api.postressource(10).put({'title':'un test'})
True
>>>api.postressource(10).get()
{u'status': 2, u'content': u'ici le contenu de mon article', u'allow_comments':
True, u'description': u'ici le contenu de mon article', u'rating_count': 0,
u'rating_average': 0.0, u'slug': u'mon-nouvel-article', u'title': u'un test',
u'keywords_string': '', u'_meta_title': None, u'expiry_date': None,
u'comments_count': 0, u'featured_image': None, u'short_url': None,
u'publish_date': u'2012-10-09T07:56:25.779764+00:00', u'id': u'10',
u'gen_description': True, u'resource_uri': u'/api/v1/postressource/10/'}
```

Votre ressource est bien modifiée.

Supprimer une ressource

Pour supprimer une ressource, le protocole HTTP a prévu le verbe `DELETE`. Son utilisation ne nécessite pas que vous précisiez un dictionnaire d'attributs. Sa syntaxe est donc très similaire à celle de `GET`. Ainsi, vous pouvez supprimer votre article de blog avec la commande suivante :

```
>>> api.postressource(10).delete()
True
```

La réponse du serveur vous indique une nouvelle fois que votre action a été menée à bien. Vous pouvez le vérifier en tentant d'atteindre la ressource avec le verbe `GET` :

```
>>> api.postressource(10).get()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/yohann/Dev/OAUTH_TEST/slumb_env/lib/python2.7/site-packages/
slumber/__init__.py", line 113, in get
    resp = self._request("GET", params=kwargs)
  File "/Users/yohann/Dev/OAUTH_TEST/slumb_env/lib/python2.7/site-packages/
slumber/__init__.py", line 104, in _request
    raise exceptions.HttpClientError("Client Error %s: %s" % (resp.status_code,
url), response=resp, content=resp.content)
slumber.exceptions.HttpClientError: Client Error 404: http://localhost:8000/
api/v1/postressource/10/
```

Cette fois, le code d'erreur qui vous est retourné devrait vous éclairer : `404 Ressource Non disponible`. C'est ce que nous souhaitions puisque vous venons de détruire la ressource.

Filtrer les ressources

Maintenant que vous maîtrisez les principaux verbes HTTP, vous souhaitez certainement en faire plus avec votre API. Un bon départ peut être d'offrir à vos clients la possibilité de filtrer les ressources selon certains critères. Pour cela, vous allez à nouveau devoir modifier votre API. Il vous faut également apprendre à filtrer les ressources côté client. Vous allez le voir, la façon de faire est assez proche de ce que vous connaissez déjà avec les querysets.

Tout d'abord, avant de pouvoir mettre en place des filtres, il faut savoir quels sont les champs sur lesquels vous allez les appliquer. Ici, l'idée est de se placer côté client. En effet, si évidemment vous pouvez avoir la liste des champs en regardant simplement les modèles du blog de Mezzanine, vous n'aurez pas toujours cette possibilité dans un contexte réel. Du côté du client, donc, pour savoir quels sont les champs disponibles,

le mieux reste encore de récupérer un article au hasard et d'itérer sur les clés de la réponse, qui est un dictionnaire. Par exemple :

```
>>> article = api.postressource(5).get()
>>> for k in article.iterkeys():
>>>     print k
status
content
allow_comments
description
rating_count
rating_average
slug
title
keywords_string
_meta_title
expiry_date
comments_count
featured_image
short_url
publish_date
id
gen_description
resource_uri
```

Vous avez donc maintenant une liste de champs à votre disposition. Commencez par un test dans le navigateur – en remplaçant l'`access_token` par le vôtre, bien entendu :

```
http://localhost:8000/api/v1/postressource/
?format=json&access_token=8bc1ba0197e53093dfda807ed0dbe48e2f6571a5&title__start
swith=django
```

La réponse du serveur, `The 'title' field does not allow filtering`, vous enseigne deux choses.

- Vous avez visiblement utilisé une syntaxe correcte.
- Votre API n'autorise pas l'utilisation des filtres.

En effet, `tastypie`, votre générateur d'API, doit d'abord être configuré, ressource par ressource, afin de proposer un système de filtre. Cette configuration est en revanche très simple. Prenez votre fichier `blog_api/api.py`, dans lequel la classe `PostRessource` contient une sous-classe (en langage Python, on parle d'inner class) `Meta` qui permet de la configurer.

Pour le moment, la classe `Meta` de `PostRessource` ressemble à ceci :

```
    class Meta:
        queryset = BlogPost.objects.all()
        detail_allowed_methods = ['get', 'post', 'put', 'delete']
        authentication = Oauth2Authentication()
        authorization = Oauth2Authorization()
```

Vous avez donc précisé pour le moment :

- une queryset à utiliser ;
- les méthodes ou verbes que vous autorisez sur votre API ;
- une méthode d'authentification (ici, OAuth2) ;
- une méthode d'autorisation (ici, une limitation aux ressources de l'utilisateur).

Vous n'avez donc plus qu'à ajouter des filtres, qui prennent la forme d'un dictionnaire :

```
filtering = {
            'title': ['startswith']
            }
```

Ici, nous avons précisé le champ sur lequel appliquer le filtre, `title`, et le type de filtre, `'startswith'`. Cette façon de faire est la plus restrictive qui soit. Nous pourrons, au fur et à mesure de nos besoins, ajouter des mots-clés dans la liste et écrire par exemple :

```
filtering = {
            'title': ['startswith', 'istartswith', 'exact', 'iexact']
            }
```

pour laisser plus d'options de filtrage au client.

Enfin, si nous souhaitons donner un accès complet aux filtres, il faut tout d'abord importer la constante `ALL` de `tastypie` :

```
from tastypie.constants import ALL
```

Ensuite, vous pouvez écrire :

```
filtering = {
            'title': ALL
            }
```

La syntaxe des filtres est exactement identique à celle des querysets. Ainsi, vous avez à votre disposition l'ensemble des filtres que Django propose dans ses querysets.

Conclusion

Le développement web s'oriente de plus en plus vers l'interconnexion de services au travers d'API REST. Ainsi, grâce à un langage commun basé sur le protocole HTTP compris par tous les langages, vous pouvez facilement maîtriser les communications entre des applications qui ont hébergées sur de multiples machines et dont les langages sont différents. Lorsque vous développez une application Django, il est très simple d'ajouter ces fonctionnalités et de proposer ce type d'interface.

15

Les contribs Django

Les « contribs » sont des modules optionnels sans lesquels Django peut tout à fait fonctionner. Cependant, ils sont souvent très utiles dans le cadre du développement d'une application web. Explorons-les et penchons-nous notamment sur les plus importants, `auth` *et* `admin`*, mais également sur* `comments`, `contenttypes`, `flatpages`, `message`, `sitemaps`, `site` *et* `syndication`.

Le module auth : authentification des utilisateurs

Certainement le plus utilisé des modules, `auth` sert à gérer l'authentification des utilisateurs. Basé sur une architecture modulaire et puissante, il gère non seulement les mécaniques habituelles des utilisateurs (connexion, déconnexion, vérification de connexion, changement de mot de passe, mot de passe oublié…), mais il propose aussi une gestion très fine des droits d'accès. Si vous êtes habitué à la logique Unix de la gestion des utilisateurs, vous retrouverez dans ce module des notions qui vous sont familières comme les groupes et les niveaux de permission.

Installation

L'installation mérite que l'on s'y intéresse, même si cela reste très simple. En effet, ce module offre des options de configuration souvent ignorées alors qu'elles peuvent vous simplifier énormément la vie.

Tout d'abord, lorsque vous créez un projet avec la commande :

```
python manage.py startproject monprojet
```

Django active par défaut le système d'authentification. Cela signifie que dans le cas où vous souhaitez utiliser ce système sans modifications particulières, il est immédiatement fonctionnel. Pour désactiver le module `auth`, il faut donc supprimer la ligne suivantes des `INSTALLED_APPS` dans `settings.py` :

```
django.contrib.auth
```

Dans le cas où vous auriez besoin de le réactiver, il vous faudra également ajouter `django.contrib.contenttypes` dont `django.contrib.auth` dépend :

```
INSTALLED_APPS = [
django.contrib.contenttypes,
django.contrib.auth,
vos_autres_applications...
]
```

Choix du backend

Vu qu'il est très simple de définir son propre *backend* d'authentification, Django n'en propose que deux par défaut :

- `ModelBackend`, qui est le plus couramment utilisé – c'est celui dont nous allons parler tout au long de ce chapitre ;
- `RemoteUserBackend`, dont le travail consiste à déléguer l'authentification à un autre service.

`ModelBackend` définit la manière dont on teste qu'un utilisateur est connecté, renvoie un utilisateur quand cela est demandé par les autres parties du système, retrouve toutes les permissions d'un groupe ou d'un utilisateur et vérifie si un utilisateur possède la permission demandée. Dans le cas de grandes entreprises qui vont gérer leurs utilisateurs au travers de l'authentification Apache par exemple, cela peut être très pratique car vous n'avez pas besoin de créer un système à part. Vous pouvez tout à fait utiliser votre ancien système d'authentification et simplement l'indiquer à Django par l'utilisation de `RemoteUserBackend`.

RemoteUserBackend

Pour utiliser `RemoteUserBackend`, vous devez activer certains middlewares de Django :

```
django.contrib.auth.middleware.AuthenticationMiddleware
```

et

```
django.contrib.auth.middleware.RemoteUserMiddleware
```

Pour cela, écrivez simplement les lignes suivantes dans votre `settings.py` :

```
MIDDLEWARE_CLASSES = (
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.RemoteUserMiddleware',
    ...
    )
```

Vous devez ensuite préciser à Django d'utiliser `RemoteUserBackend` au lieu du plus classique `ModelBackend`. Toujours dans `settings.py`, écrivez :

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.RemoteUserBackend',
)
```

Les middlewares

Les middlewares sont un système bas niveau de gestion du processus `request/response`. Ils interceptent la `request` à différents moments de sa vie et la modifient ou effectuent certaines actions en fonction de certains critères. De la même manière, ils peuvent intercepter la `response` avant qu'elle ne soit livrée au navigateur client.

Par la suite, Django s'attend à trouver dans le dictionnaire `META` de chaque requête une clé `REMOTE_USER` contenant le nom de l'utilisateur.

`RemoteUserBackend` est bien entendu pleinement configurable, via le surclassage de la classe initiale. C'est aussi un bon exemple pour découvrir comment créer de nouveaux backends. Avant cela, nous allons nous plonger dans les fonctionnalités du module `auth`. Ce sera ensuite beaucoup plus simple pour vous de comprendre ce que vous avez besoin de faire pour implémenter votre propre système d'authentification.

L'objet User

Le système d'authentification de Django repose beaucoup sur l'objet `User`, un modèle que vous retrouverez dans `django.contrib.auth.models`. Il renferme toutes les informations nécessaires pour faire fonctionner l'authentification. Les champs les plus importants sont `username`, qui correspond au surnom de l'utilisateur, et `password`, qui renferme son mot de passe.

La sécurité des mots de passe

Vous avez sans doute déjà entendu parler de sites de renom qui avaient été victimes d'attaque et dont les comptes utilisateurs avaient été compromis. Cela tient au fait que sur ces sites, les couples nom d'utilisateur/mot de passe étaient écrits en langage clair et non cryptés dans la base de données. Ainsi, un accès à la base de données de ces sites permettait d'accéder aux informations confidentielles des utilisateurs. Django, lui, utilise un système de cryptage des mots de passe qui évite ce type de mésaventures, via le système PBKDF2 par défaut – vous pouvez employer votre propre système si vous le souhaitez.

Création d'utilisateur

La première chose à faire est de créer de nouveaux utilisateurs. Pour cela, voyons comment faire grâce à la ligne de commande :

```
>>> from django.contrib.auth.models import User
# Vous commencez donc par importer le modèle User dont vous aurez besoin.
>>> user = User.objects.create_user('bob','test@example.com', 'mot de passe')
```

Cet exemple de code est très simple, mais il mérite quelques explications. Nous n'avons pas employé la syntaxe habituelle :

```
>>> user = User(username='bob', email='test@exemple.com', password='mot de
passe')
```

Nous sommes passés par la méthode `create_user` du manager `objects` de la classe `User`. Entre autres choses, cette méthode va crypter le mot de passe. Ainsi, si l'on cherche à obtenir le mot de passe de Bob, on n'obtiendra que sa version cryptée :

```
>>> user.password
'pbkdf2_sha256$10000$YINDgWLIyVj7$JddeZBmSt2xGBL2Mv7UDK6k/xVUXwBjvvUISPOI8e4Y='
```

Si vous aviez utilisé la méthode traditionnelle de création d'objet, Django aurait enregistré « mot de passe » dans le champ `password` de la base de données.

Si vous essayez chez vous, avec les mêmes informations, vous obtiendrez une version cryptée différente. Cela tient au fait que Django emploie dans son calcul de mot de passe la variable `SECRET_KEY` présente dans `settings.py`. Au moment de vérifier le mot de passe, Django décode la chaîne de caractères enregistrée dans la base de données en s'aidant de cette variable `SECRET_KEY`. Dans le cas où vous avez utilisé `create_user`, le bon mot de passe sera décodé. En revanche, si vous employez la méthode traditionnelle, cela va échouer. C'est donc essentiellement pour des raisons de sécurité que vous devez toujours utiliser `create_user` pour la création de nouveaux utilisateurs. Enfin, vous remarquerez que nous ne faisons pas appel à la méthode `save`, car `create_user` s'en charge pour nous.

Activer un utilisateur

Après cette étape de création, l'utilisateur n'est pas encore utilisable (fonctions de connexion et de déconnexion indisponibles, par exemple). Pour y remédier, vous devez faire en sorte que l'utilisateur soit considéré comme « actif » par le système :

```
>>> user.is_active = True
>>> user.save()
```

Votre nouvel utilisateur est maintenant en mesure de se connecter via les méthodes prévues à cet effet, mais pourquoi cette étape supplémentaire ? Nous pourrions imaginer un système où les utilisateurs sont immédiatement actifs sitôt leur création. Seulement, l'ajout d'une étape supplémentaire sert à couvrir un cas d'utilisation extrêmement fréquent. Cela vous est sans doute déjà arrivé plusieurs fois : vous vous enregistrez sur un site Internet où vous sélectionnez un nom d'utilisateur et un mot de passe. Vous fournissez ensuite une adresse électronique. Puis le site vous envoie un e-mail contenant un lien pour activer votre compte ; cela lui permet de valider l'existence de l'adresse de contact fournie. Avec `is_active`, vous mettez très facilement en place ce système d'enregistrement. C'est d'ailleurs ce que propose `django-registration`, un module Python pour Django spécialement dévolu à cette tâche (https://bitbucket.org/ubernostrum/django-registration/).

Un second cas d'utilisation concerne, cette fois, la suppression des comptes. Si, par exemple, pour une raison ou pour une autre, vous ne souhaitez plus autoriser l'un des auteurs à publier sur votre blog, vous pouvez déactiver le compte sans pour autant supprimer les articles qu'il a déjà écrits.

Connecter un utilisateur

Vous avez donc maintenant un compte utilisateur qui est à la fois créé et activé. Il est temps de lui laisser la possibilité de se connecter sur votre application.

Utiliser authenticate() et login()

L'application `auth` de Django offre deux fonctions pour la connexion : `login()` et `authenticate()`. Attention, vous devez d'abord utiliser `authenticate()` avant `login()`. La première vous renvoie un objet `User` et la seconde enregistre l'utilisateur dans la session. Pour des besoins didactiques, nous emploierons directement le dictionnaire `POST` fourni par l'utilisateur. En temps normal, vous auriez créé un formulaire et vous utiliseriez donc `cleaned_data`.

```
from django.contrib.auth import authenticate, login
def ma_vue(request) :
    login = request.POST['login']
```

```
    mot_de_passe = request.POST['mot_de_passe']
    user = authenticate(login, mot_de_passe)
```

À partir de maintenant, la variable `user` correspond soit à `None` si l'authentification a échoué, soit à un objet de type `User` si elle s'est déroulée avec succès. Il est possible de le vérifier :

```
if user:
    # Code exécuté si l'utilisateur a été authentifié avec succès.
else :
    # Renvoyez un message d'erreur à l'utilisateur.
```

Il faut ensuite vérifier que cet utilisateur authentifié est actif :

```
if user.is_active :
    # Ici, l'utilisateur est authentifié et son compte est actif.
else :
    # Ici, le compte est désactivé.
```

Si le test échoue, il faut renvoyer un message à l'utilisateur lui expliquant que son compte est suspendu, en lui fournissant s'il y a lieu une méthode d'activation (suivre un lien d'activation dans un e-mail, contacter le service d'assistance…).

Vous voici maintenant à l'étape où le compte est à la fois authentifié et actif. Il reste maintenant à enregistrer cette information dans la session. Django offre une méthode simplifiée pour le faire et vous épargne d'avoir à jouer avec l'objet `session` :

```
login(request, user)
```

Vous pouvez renvoyer l'utilisateur sur une page représentant son profil par exemple. Vous avez maintenant dans la `request` l'objet `User` qui représente l'utilisateur. Vous pouvez donc écrire dans votre template :

```
Bonjour {{user.username}}
```

Voici maintenant le code dans son ensemble :

```
def ma_vue(request):
    login = request.POST['login']
    mot_de_passe = request.POST['mot_de_passe']
    user = authenticate(login, mot_de_passe)
    if user is not None:
        if user.is_active:
            login(request, user)
```

```
        else :
            # Le compte est désactivé.
    else:
        # L'authentification a échoué.
```

Utiliser views.login()

La vue que vous venez d'écrire est fastidieuse et répétitive. Dès que vous aurez besoin de connecter un utilisateur sur votre site, vous devrez employer une vue de ce type. Et de plus, nous n'avons même pas traité le formulaire. Heureusement, comme pour tout ce qui est rébarbatif et répétitif, Django propose une solution, qui réside dans les vues de l'application `auth`.

La vue `views.login`, notamment, a le comportement suivant : si elle est appelée avec le verbe `GET`, elle affiche un formulaire et si c'est avec le verbe `POST`, elle tente d'authentifier l'utilisateur, de le connecter et de le renvoyer sur une page que vous aurez choisie. Pour cela, vous devez renseigner la variable `LOGIN_REDIRECT_URL` dans le fichier `settings.py`. Si cette page n'est pas renseignée, Django renverra l'utilisateur sur l'URL `/accounts/profile`. Vous aurez également besoin d'un template représentant le formulaire de login. Par défaut, Django ira le chercher dans `registration/login.html`, mais vous pourrez changer cela si ça ne vous convient pas. Voici à titre d'exemple ce à quoi un formulaire de login pourrait ressembler :

```
<form method="post" action="{% url 'django.contrib.auth.views.login' %}">
{% csrf_token %}
{{ form.as_ul }}
<input type='submit' value='se connecter' />
<input type='hidden' value={{next}}
</form>
```

Remarquez le champ caché dont la valeur est `next`, qui va rediriger l'utilisateur non sur la page prévue, mais sur celle qu'il souhaitait atteindre. Vous serez confronté à ce problème lorsque vous allez restreindre certaines de vos pages aux seuls utilisateurs enregistrés. Dans le cas où un utilisateur souhaite atteindre l'une de ces pages, il sera redirigé vers le formulaire de login avant de pouvoir y accéder. Une fois qu'il se sera connecté, il ne sera pas redirigé vers sa page de profil par exemple, mais directement vers celle qu'il souhaitait atteindre au départ.

Maintenant que tout est préparé (un template contenant le formulaire et une URL pour rediriger l'utilisateur après qu'il se soit logué), il ne vous reste plus qu'à renseigner l'URL de login dans votre fichier `urls.py` :

```
(r'^se_connecter/'django.contrib.auth.views.login'),
```

Si vous souhaitez utiliser un autre template que celui par défaut, vous pouvez écrire :

```
(r'^se_connecter/', 'django.contrib.auth.views.login', {'template_name':
'votre_template_de_login.html'}),
```

Déconnecter un utilisateur

Maintenant que vous savez connecter un utilisateur, il est utile d'apprendre à le déconnecter. Là encore, il existe deux possibilités : soit le faire manuellement, soit utiliser une vue prévue par Django.

Utiliser logout()

Contrairement à `login()`, vous n'avez pas de travail préalable à faire pour déconnecter un utilisateur. Votre vue va donc être vraiment très simple :

```
def deconnexion(request) :
    logout(request)
    # Redirigez votre utilisateur sur l'URL de connexion par exemple.
```

Utiliser views.logout()

Il est également possible d'employer la vue `logout()` de `contrib.auth.views`. Sans configuration particulière, cette vue retourne le template `registration/logged_out.html`. Dans ce template, vous trouverez le contexte suivant :

- l'objet `site` et la variable `site.name` avec l'application contrib `site` ;
- la variable `title` correspondant à la version internationalisée de `Logged Out` (déconnecté).

Si vous choisissez d'écrire le template `registration/logged_out.html`, vous n'aurez pas besoin d'en faire plus et votre URL devrait ressembler à ceci :

```
(r'^se_deconnecter/', 'django.contrib.auth.views.logout'),
```

Bien évidemment, il est possible de changer la localisation du template :

```
(r'^se_deconnecter/', 'django.contrib.auth.views.logout', {'template_name' :
'votre_template_de_deconnexion.html'}),
```

et de rediriger l'utilisateur vers une page de votre choix, par exemple la page d'accueil de votre site :

```
(r'^se_deconnecter/', 'django.contrib.auth.views.logout', {'next_page' :
'home'}),
```

Enfin, vous pouvez rediriger l'utilisateur sur votre page de login plutôt que sur la page d'accueil. Dans ce cas, il est possible d'employer une autre vue que vous propose le module `auth` : `logout_then_login`. L'avantage de cette vue tient en sa concision. En effet, si vous avez indiqué votre `LOGIN_URL` dans `settings.py`, vous n'avez plus qu'à écrire :

```
(r'^se_deconnecter/', 'django.contrib.auth.views.login_then_logout'),
```

Dans le cas contraire, vous avez bien entendu toujours la possibilité de le faire manuellement :

```
(r'^se_deconnecter/', 'django.contrib.auth.views.login_then_logout',
{'login_url', 'votre_url_de_login'),
```

Vérifier qu'un utilisateur est connecté

Vous allez être confronté à deux cas courants d'utilisation.

- Vous limiterez l'accès à votre site aux seuls utilisateurs connectés. Pour cela, vous n'emploierez pas les données de l'objet `User`, mais vérifierez simplement que l'utilisateur est bien connecté.
- Vous afficherez à l'utilisateur une page le concernant, par exemple une page de profil.

Au niveau de l'implémentation Django, l'un et l'autre cas sont perçus de la même manière. Vous devez simplement vous assurer que l'utilisateur est connecté : soit directement dans votre code, soit en utilisant un décorateur sur vos vues, soit en plaçant un décorateur sur vos URL.

La méthode manuelle

L'objet `User` possède une méthode indiquant si l'utilisateur est connecté ou non. Et vous avez toujours à disposition un objet `user` dans l'objet `request`. Seulement, dans le cas où aucun utilisateur n'est connecté, un utilisateur un peu spécial est employé : `Anonymous`. Plutôt que de chercher à vérifier qu'il s'agit bien de cet utilisateur, vous pouvez employer `is_authenticated` de la manière suivante :

```
if request.user.is_authenticated() :
    # L'utilisateur est bien connecté, vous pouvez continuer à traiter votre
vue.
else :
    # L'utilisateur n'est pas connecté, vous pouvez le rediriger vers la page de
login.
```

Le décorateur sur les vues

Dans le cadre d'un usage courant, il est plus simple d'employer un décorateur sur votre vue. En effet, le code est plus concis et Django émettra des assertions qui correspondront presque toujours à vos besoins. Ce décorateur part du principe que :

- si l'utilisateur n'est pas authentifié, il faut le rediriger sur la page de login ;
- il faut ajouter le paramètre `next` à l'URL pour rediriger l'utilisateur vers la vue qu'il tentait d'atteindre avant d'être forcé de se connecter.

Voici comment s'en servir :

```
from django.contrib.auth.decorators import login_required
@login_required
def ma_vue(request) :
    # Ici, vous pouvez utiliser l'objet request.user en étant certain qu'il
s'agit bien d'un utilisateur connecté.
```

Quand l'utilisateur n'est pas authentifié, il est redirigé vers l'URL spécifiée dans la variable `LOGIN_URL` de `settings.py`. Ce comportement est à la fois simple et pratique. Pourtant, vous souhaiterez parfois déplacer la logique d'authentification à un niveau légèrement supérieur, dans les URL par exemple.

Le décorateur dans les URL

Il est parfaitement admis d'utiliser les URL pour y placer vos décorateurs. Cette façon de faire est même un peu plus DRY, puisqu'une simple lecture des URL indique ce qui est protégé et ce qui ne l'est pas. De plus, c'est la seule façon de limiter l'accès aux vues basées sur des classes. Voici comment s'y prendre :

```
from django.contrib.auth.decorators import login_required

from monapp.views import ma_vue

urlpatterns = patterns('',
    (r'^ma_vue/', login_required(ma_vue)),
```

La gestion des utilisateurs connectés dans les templates

Un cas pratique assez courant consiste à afficher un contenu spécifique selon que l'utilisateur soit connecté ou non. Imaginons, par exemple, que sur le menu de votre site vous souhaitiez afficher le nom de l'utilisateur s'il est connecté ou un lien vers le formulaire de login dans le cas contraire. Pour cela, ce n'est pas au niveau de la vue que vous devez implémenter cette logique, mais directement dans le template.

Comme le template a accès à l'objet `User`, si vous avez renvoyé une vue contenant le contexte, vous pouvez faire un test simple avec `is_authenticated` :

```
{% if user.is_authenticated %}
Hello {{user.username}}
{% else %}
<a href='{% url login %}'>Se connecter</a>
{% endif %}
```

À partir de maintenant, vous avez un système d'authentification robuste et complet, en ce qui concerne le minimum que l'on demande à un système d'authentification, c'est-à-dire connecter et déconnecter un utilisateur ainsi que valider qu'il est bien connecté pour accéder à une URL. Néanmoins, vous pouvez en faire bien plus avec le système d'authentification de Django, en particulier concernant les permissions et les groupes.

Les permissions

Les permissions servent à gérer plus finement les autorisations sur le site. Pour le moment, vous n'avez qu'un système binaire : utilisateur connecté ou non. Avec un système de permissions, vous pouvez gérer des problèmes du type « qui fait quoi sur quoi » ; vous avez donc trois intervenants dans un système de permissions :

- le « qui » est l'utilisateur ;
- le « fait quoi » est l'action à autoriser ou non ;
- le « sur quoi » est habituellement un modèle Django.

Les permissions que vous allez traiter sont de trois types : ajouter, modifier et supprimer. Elles sont liées à un modèle et créées automatiquement par Django pour chaque modèle de chaque application. Il n'existe pas de permission « lire » car, même s'il n'y a pas de modification, Django considère qu'il n'y a pas d'intérêt à voir la liste des objets si l'on ne peut les modifier.

Les permissions d'un utilisateur

Django ne précise pas de permissions par défaut à la création d'un utilisateur. C'est à vous de définir au cas par cas les permissions des utilisateurs sur les modèles de votre choix. Cela signifie qu'un utilisateur nouvellement créé n'a tout simplement accès à rien.

> **CAS PARTICULIER Le super-admin**
>
> Lorsque vous créez un nouvel utilisateur, vous avez la possibilité de positionner `is_super_user` à `True`. Dans ce cas, l'utilisateur en question a par défaut accès à tout.

Ajouter ou supprimer des permissions

Pour vérifier les permissions d'un utilisateur, vous devez utiliser le champ `user_permissions`. Testons cette méthode sur un utilisateur nouvellement créé, qui ne possède donc encore aucune permission :

```
>>> user = User.objects.create_user('test', 'test@example.com', 'mot_de_passe')
>>> user.is_active = True
>>> user.save()
>>> user.user_permissions.all()
[]
```

Ajoutons à cet utilisateur la permission de créer des entrées de logs dans l'application `admin` :

```
from django.contrib.auth.models import Permission
perm = Permission.objects.get(codename='add_logentry')
user.user_permissions.add(perm)
user.save()
```

Dans un premier temps, après avoir importé le modèle `Permission`, nous retrouvons la permission qui nous intéresse grâce à son `codename`. Le *codename* des permissions automatiques de Django est composé du mot-clé `add`, `change` ou `delete` et du nom du modèle en minuscules séparé par un _. Nous utilisons ensuite la syntaxe classique des ajouts sur une relation many-to-many, puis nous sauvegardons l'objet `User`. Comme `user_permissions` est une relation many-to-many, vous pouvez également supprimer une permission :

```
user.user_permissions.remove(perm)
```

ou toutes les permissions d'un modèle :

```
user.user_permissions.clear()
```

Vérifier une permission

L'objet `User` propose la méthode `has_perm` qui prend en argument une chaîne de caractères et renvoie `True` si l'utilisateur possède la permission ou `False` dans le cas contraire. Cette chaîne de caractères doit être composée du nom de l'application suivi du codename de la permission. Dans notre exemple, `add_logentry` est une permission de l'application `admin`. Pour la vérifier, il faut donc écrire :

```
>>> user.has_perm('admin.add_logentry')
True
```

Vous pouvez donc employer cette méthode pour vérifier une permission dans vos vues. Comme pour vérifier si un utilisateur est logué, le module `auth` vous propose un décorateur à placer sur vos vues pour simplifier cette vérification.

permission_required

Le décorateur `permission_required` agit de manière très similaire à `login_required` :

```
from django.contrib.auth.decorators import permission_required

@permission_required('admin.add_logentry')
def ma_vue(request) :
    # le reste du code de votre vue
```

Par défaut, ce décorateur va renvoyer l'utilisateur sur l'URL de `LOGIN_URL` que vous avez définie dans `settings.py`. Pourtant, il est possible que ce ne soit pas ce que vous désiriez. Dans ce cas, ajoutez un paramètre supplémentaire à votre décorateur, `login_url`. Si l'utilisateur n'a pas les permissions demandées, il sera redirigé vers l'URL que vous venez de définir :

```
@permission_required('admin.add_logentry', login_url='/home/')
def ma vue(request) :
    # le reste du code de votre vue
```

Les groupes

Le système de permissions que nous venons de voir, même s'il est souple et simple d'usage, manque d'une fonctionnalité très importante : la gestion des groupes. La définition d'un groupe consiste à attribuer des permissions similaires à un ensemble d'utilisateurs pour un certain type d'utilisation de votre application. Ainsi, dans le cadre d'un journal en ligne par exemple, vous allez définir des auteurs et des relecteurs. Les premiers doivent pouvoir créer, modifier et supprimer des articles et les seconds ne doivent être autorisés qu'à les modifier. Bien entendu, vous pouvez ajouter les bonnes permissions à chaque auteur créé, puis faire de même pour les relecteurs, mais avouez que c'est rapidement fastidieux et sujet aux erreurs humaines.

C'est pour ce type de cas que le système de groupes de Django est particulièrement intéressant. Il va attacher des permissions non plus à un utilisateur mais à un objet de type `Group`, qui sera ensuite attaché à un utilisateur. Dans le cas des auteurs et des relecteurs, c'est particulièrement simple. Vous allez une première fois définir des permissions pour les auteurs et d'autres permissions pour les relecteurs, et vous n'aurez plus ensuite qu'à placer un utilisateur dans l'un ou l'autre groupe.

Créer un groupe

Un `Group` est un simple modèle Django qui ne nécessite aucune configuration particulière, à part que vous lui donniez un nom :

```
from django.contrib.auth.models import Group
Group.objects.create(name="auteurs")
Group.objects.create(name="relecteurs")
```

Vos deux groupes sont maintenant créés.

Ajouter une permission à un groupe

Il vous faut encore leur ajouter les bonnes permissions :

```
from django.contrib.auth.models import Group, Permission
auteurs = Group.objects.get(name="auteurs")
auteurs.permissions =
list(Permission.objects.filter(codename__endswith='article'))
```

Dans cet exemple, plutôt que d'ajouter les permissions une à une, nous avons évalué une queryset, avec :

```
list(Permission.objects.filter(codename__endswith='article'))
```

et nous l'avons affectée à `auteurs.permissions`. C'est plus rapide que d'écrire le code suivant :

```
for perm in Permission.objects.filter(codename__endswith='article') :
    auteurs.permissions.add(perm)
```

Les relecteurs n'ont que la permission de modifier un article. Vous pouvez donc utiliser la syntaxe suivante :

```
relecteurs = Group.objects.get(name='relecteurs')
relecteurs.permissions.add('change_article')
```

Ajouter un utilisateur à un groupe

Maintenant que les groupes sont dotés de leurs permissions, il faut encore les lier aux utilisateurs :

```
from django.contrib.auth.models import User, Group
jean = User.objects.get('username' = 'jean')
jean.groups.add(relecteurs)
```

Vous pouvez ensuite tester les permissions de l'utilisateur de la même manière que pour un utilisateur unique. En effet, Django ne fait aucune différence entre une permission qui a été octroyée via un groupe ou directement à l'utilisateur.

Créer ses propres permissions

Le système de permissions par défaut est simple et pratique, mais il ne correspond pas toujours à ce que nous souhaitons. Par exemple, dans le journal en ligne, un auteur ne devrait pas avoir la possibilité de publier un article ; seul un relecteur devrait y être autorisé. Seulement, il n'existe pas de permission « publier ». C'est pourquoi le module `auth` vous offre la possibilité de définir vous-même vos permissions : elles se comporteront en tout point comme celles que Django définit pour vous. Ainsi, elles se trouvent dans la liste des permissions disponibles sur la fiche d'un utilisateur dans l'`admin`. Les permissions supplémentaires se définissent dans les métas d'un modèle. Il faut leur donner un codename et une description.

```
class Article(models.Model) :

    class Meta :
        permissions = (
        ('publier', 'peut publier un article'),
        )
```

Pour que cette nouvelle permission soit créée par Django, vous devez faire un `syncdb` avant de l'utiliser :

```
python manage.py syncdb
```

Ensuite, vérifiez qu'un utilisateur possède bien la permission adéquate :

```
user.has_perm('blog.publier_article')
```

En bref

L'application `auth` est à la fois souple et simple d'utilisation. Elle répond à un besoin très courant dans les applications web et il existe de nombreux modèles qui en font usage. N'hésitez donc pas à en tirer pleinement parti dans vos développements.

Le module admin : gérer l'interface d'administration de Django

Le module `admin` est certainement celui que vous utiliserez le plus souvent avec le module `auth`. C'est lui qui se charge de vous présenter l'interface d'administration de Django. Comme vous en avez déjà fait grand usage au fil des applications que vous avez construites, nous allons découvrir comment rendre cette interface d'administration plus conviviale, comment la personnaliser en profondeur et surtout comment faire en sorte qu'elle ne joue pas contre vous mais avec vous.

Cas d'utilisation du module admin

Il existe trois cas d'utilisation différents du module `admin` : prototypage, maintenance et interface pour les usagers.

Le prototypage

Pour le prototypage de votre application, vous utilisez le module `admin` simplement pour vérifier que vos modèles se comportent bien comme vous l'escomptiez. L'interface d'administration sert à créer aisément quelques objets, à tester les relations entre eux, et à valider que votre architecture de base de données correspond bien à vos souhaits. Dans ce premier cas, l'interface d'administration ne sera pas mise en production ; vous n'avez donc pas à vous préoccuper de son aspect graphique et vous pouvez mettre en place quelques astuces pour prototyper rapidement votre application.

Comment activer l'interface d'administration

Pour mettre en place l'interface d'administration, décommentez la ligne suivante dans les `INSTALLED_APPS` de `settings.py` :

```
'django.contrib.admin',
```

Sachez que pour fonctionner normalement, l'interface d'administration nécessite également (toujours dans `INSTALLED_APPS`) :

```
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
```

Pour que l'interface soit fonctionnelle, il faut créer les modèles dont Django a besoin :

```
python manage.py syncdb
```

Ensuite, vous devez connecter une URL à l'interface d'administration. Dans le fichier racine de vos URL, décommentez :

```
from django.contrib import admin
admin.autodiscover()
```

puis décommentez :

```
url(r'^admin/', include(admin.site.urls))
```

Notez que le fait que l'interface d'administration soit disponible à l'adresse `/admin/` est une convention ; il est donc tout à fait possible de choisir une autre URL.

Enfin, dans les applications qui vont utiliser l'interface d'administration, vous devez créer un fichier `admin.py`, dont la syntaxe basique est la suivante :

```
from django.contrib import admin
from monapplication.models import MonModel
admin.site.register(MonModel)
```

Relancez ensuite votre serveur de test :

```
python manage.py runserver
```

Rendez-vous dans votre interface d'administration avec votre navigateur, à l'adresse `http://localhost:8000/admin/`.

L'interface d'administration de prototypage

Dans le cas d'un prototypage, la mise en place de l'interface d'administration doit vous prendre le moins de temps possible.

Pour cela, vous pouvez tout à fait réduire la syntaxe du fichier `admin.py`. Vous avez appris qu'il faut écrire ce qui suit :

```
from django.contrib import admin
from monapplication.models import MonModel, MonAutreModel, EncoreUnAutreModel
admin.site.register(MonModel)
admin.site.register(MonAutreModel)
admin.site.register(EncoreUnautreModel)
```

Cependant, si vous n'avez pas de besoins particuliers, n'hésitez pas à utiliser la concision de Python ; cela fonctionne tout aussi bien et fait gagner quelques lignes :

```
from django.contrib import admin
from monapplication.models import MonModel, MonAutreModel, EncoreUnAutreModel

for model in [MonModel, MonAutreModel, EncoreUnAutreModel] :
    admin.site.register(model)
```

Afin que vos modèles soient lisibles dans l'interface d'administration, vous devez définir une méthode `__unicode__`. Or, il peut être fastidieux de le faire pour chacun de vos modèles. Là encore, utilisez la souplesse de Django et de Python réunis et optez pour une classe abstraite. Dans les modèles, une telle classe ne crée pas de tables en base de données. Si vous définissez des champs dans cette classe, ils sont ajoutés aux classes filles, c'est-à-dire celles qui vont hériter de votre classe abstraite. En revanche, si vous ne définissez que des méthodes, c'est absolument sans conséquence pour votre base de données. Vous pouvez donc écrire une classe abstraite qui ne contiendra que des méthodes et qui ne sera employée que lors du prototypage.

Si, par exemple, vos modèles s'identifient majoritairement par le champ `name`, vous pouvez écrire :

```
class Prototypage(models.Model) :
     class Meta :
        abstract = True
    def __unicode__(self) :
        return self.name
```

Vous n'aurez plus qu'à faire hériter vos classes de prototypage pour que la méthode `__unicode__` soit disponible pour tous vos modèles. Si l'un de vos modèles ne peut avoir le champ `name` pour identifiant, utilisez dans ce cas les possibilités du surclassement en redéfinissant dans la classe fille une méthode `__unicode__`.

Des héritages différents pour la production et le développement

Au moment du développement, si vous définissez un ensemble de méthodes que vous ne souhaitez pas utiliser en production, vous pouvez tout à fait profiter de la souplesse de Django et faire des imports conditionnels :

```
from django.conf import settings
if settings.DEBUG = True :
    from developpement.models import BaseModel
else
    from production.models import BaseModel
```

Ainsi, si votre application est en mode `debug`, vous utiliserez la classe `BaseModel` que vous avez définie dans le fichier `developpement.models`. Sinon, c'est `production.models` qui sera employé. Prenez garde néanmoins à ce que `BaseModel` soit bien une classe abstraite et quelle ne contienne que des méthodes.

Bien entendu, ces quelques exemples n'ont qu'un seul but : vous donner envie de trouver vous-même vos techniques de prototypage rapide suivant l'application que vous avez à développer.

L'interface d'administration à usage de maintenance

Dans ce cas d'utilisation, l'interface d'administration n'est pas servie au public. Seuls votre équipe de développeurs et vous-même l'utiliserez pour faire fonctionner le site Internet (pour contrôler le nombre d'utilisateurs enregistrés, faire de la modération de commentaires, vérifier que tout fonctionne comme prévu…). L'aspect graphique n'est alors pas primordial. Toutefois, vous avez tout de même besoin de fonctionnalités plus avancées que pour le prototypage. De plus, l'interface d'administration sera activée en production. Il faudra donc prendre des mesures de sécurité supplémentaires afin de ne pas avoir de mauvaises surprises.

La sécurité

Comme l'interface d'administration (au moins sa page d'accueil) sera visible des utilisateurs, vous devez vous prémunir des dangers que cela peut entraîner.

Le plus simple est de commencer par ne pas rendre l'interface d'administration disponible à l'adresse `/admin/`. En effet, comme c'est presque toujours à cette adresse que l'`admin` est servie, vous éviterez ainsi une bonne partie des attaques de *script kiddies*.

> **Les script kiddies**
>
> Cette expression anglaise péjorative qualifie un novice qui va essayer, pour le plaisir, de trouver une faille de sécurité dans votre site. Heureusement, les scripts kiddies ne savent pas vraiment ce qu'ils font et ont un niveau en sécurité informatique proche de zéro. Mais attention tout de même : ils sont assez nombreux.

Une seconde possibilité consiste à rendre inaccessible l'interface d'administration autrement qu'en HTTPS. De cette façon, tous les échanges entre le serveur et votre navigateur se feront de manière cryptée, vous garantissant une grande sécurité. Par exemple, vous pouvez créer deux projets distincts qui seront servis par deux URL différentes : `admin.monsite.com` et `monsite.com`. Le premier projet ne contiendra que l'`admin` et sera servi en HTTPS. Le second, accessible en HTTP, ne proposera que le site réservé aux visiteurs.

La documentation de l'administration

Si vous et votre équipe êtes les seuls à utiliser l'interface d'administration, il peut être intéressant de mettre en place le système de documentation intégré de Django. Ce module sert à consulter la documentation de Django et de vos applications directement dans l'interface d'administration. Une partie va être écrite pour vous par Django en s'appuyant sur vos docstrings.

Pour fonctionner, cette interface nécessite l'installation de `docutils` (http://docutils.sourceforge.net/), un module Python offrant de nombreux outils pour la documentation du code source. Installez-le via `pip`, à l'aide de la commande suivante :

```
>>> pip install docutils
```

Vous devez ensuite décommenter dans votre fichier `settings.py` :

```
'django.contrib.admindocs',
```

et dans le fichier racine de vos URL :

```
url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
```

Démarrez votre serveur local si cela n'est pas déjà fait et ouvrez votre navigateur à l'adresse `http://localhost:8000/admin/doc/`. Vous pouvez maintenant consulter la documentation de Django directement depuis votre site. Vous constaterez également que Django a créé pour vous la documentation de vos propres applications. Bien entendu, pour que la documentation de vos applications soit correctement rendue et utile, il faut que vous preniez soin de suivre les bonnes pratiques de la documentation d'un projet Python, comme expliqué dans le chapitre 3.

En plus des signes de documentation habituels pour un projet Python, vous pouvez, pour rendre votre documentation encore plus simple et lisible, en utiliser d'autres :

- `:model:'application.NomDeModele'` permet de référencer un modèle. Django va générer un lien vers la documentation du modèle ;
- `:view:'appname.view_name'` ;
- `:tag:'tagname'` ;
- `:filter:'filtername'` ;
- `:template:'path/to/template.html'`.

L'interface d'administration pour les usagers

Dans ce dernier cas d'utilisation, vous allez livrer l'interface d'administration en même temps que le reste du site. Vous devez donc veiller à soigner son ergonomie, sa

présentation, mais également sa sécurité, de manière à ce que l'utilisateur puisse s'en servir avec simplicité et plaisir.

Cacher ce qui n'est pas nécessaire

Voici un conseil plutôt général : si un utilisateur n'a pas besoin de quelque chose, cachez-le lui. Il aura ainsi une interface plus claire et vous minimiserez le risque d'erreur.

N'hésitez donc pas à créer des groupes en fonction des attributions de chacun. Chaque groupe doit avoir un nom clairement défini et un sous-ensemble de permissions qui va permettre de cacher les modules de l'`admin` dont il n'aura pas besoin. Ainsi, vous garantissez deux choses :

- vous évitez que les utilisateurs modifient par erreur des informations qui ne les concernent pas ;
- vous allégez l'interface d'administration des données inutiles.

Utilisez list_display pour afficher des informations utiles

L'attribut `list_display` correspond à la page qui liste toutes les instances d'un même modèle. L'erreur serait de considérer cette liste comme une simple représentation du modèle sous-jacent. Certaines informations sont utiles, d'autres moins. Il faut donc faire preuve de discernement pour choisir les champs du modèle qui nécessitent d'être présentés dans l'interface d'administration.

Les méthodes des modèles peuvent également vous être d'une aide précieuse pour afficher des informations qui sont utiles à l'utilisateur, mais qui ne sont pas directement liées aux modèles.

L'interface d'administration par l'exemple

Afin de maîtriser les rouages de l'`admin` et d'être en mesure de proposer aux utilisateurs une interface conviviale, nous allons étudier pas à pas un exemple. Nous montrerons les principaux avantages de l'interface et les méthodes pour éviter quelques pièges courants. Avant de commencer, assurez-vous d'avoir activé l'interface d'administration !

Présenter une liste des objets

Tout d'abord, nous allons présenter la liste des objets de vos modèles de la manière la plus simple et conviviale possible. Pour cet exemple, prenez un modèle `Profile` défini comme suit :

```
from django.contrib.auth.models import User
from django.db import models
```

```
class Profile(models.Model) :
    user = models.OneToOneField(User)
    commentaire = models.TextField()
```

Créez ensuite un fichier `admin.py`, pour l'instant très simple, dans votre application :

```
from models import Profile
from django.contrib import admin
admin.site.register(Profile)
```

La méthode __unicode__

Quand vous aurez créé plusieurs utilisateurs et plusieurs profils, l'interface d'administration va lister un ensemble de *profile objects* impossibles à différencier. Votre premier besoin consistera donc à distinguer ces objets : implémentez sur votre modèle une méthode `__unicode__` pour retrouver vos profils.

```
def __unicode__(self) :
    return self.user.username
```

Dans ce code d'une seule ligne, nous utilisons un champ d'un modèle relié et non un champ du modèle lui-même. N'hésitez pas à procéder ainsi, c'est souvent extrêmement pratique.

Rien ne vous empêche de réaliser des calculs dans la méthode `__unicode__` :

```
def __unicode__(self) :
    return '%s %s'%(self.user.first_name, self.user.last_name)
```

Dans le cas où l'utilisateur a effectivement renseigné ses nom et prénom, vous verrez son nom complet apparaître dans la liste. Dans le cas contraire en revanche, vous n'aurez qu'un champ vide sur lequel il sera impossible de cliquer. Vous devez donc affiner un peu votre méthode :

```
    def __unicode__(self):
        if self.user.first_name or self.user.last_name:
            return "%s %s" %(self.user.first_name, self.user.last_name)
        else:
            return self.user.username
```

De cette manière, vous affichez soit le nom complet de l'utilisateur, soit seulement son surnom.

Vous pouvez bien entendu aller beaucoup plus loin dans les méthodes que vous utiliserez. Sachez simplement que Django n'impose aucune limite à ce que vous pouvez imaginer.

Personnaliser l'affichage des objets

Maintenant que votre méthode `__unicode__` permet de différencier les différents profils, vous pouvez sélectionner des champs supplémentaires à afficher dans la liste. Il vous faut cette fois agir sur votre fichier `admin.py`.

Pour ajouter des options d'affichage à votre liste d'objets, vous devez créer dans ce fichier une classe qui héritera de `ModelAdmin`, qui vous donnera le contrôle de l'affichage de vos objets dans l'interface d'administration. Dans sa forme la plus simple, elle se présente comme suit :

```
class ProfileAdmin(admin.ModelAdmin) :
    pass
admin.site.register(Profile, ProfileAdmin)
```

Cette fois, la fonction `admin.site.register` prend deux arguments : le modèle que vous souhaitez personnaliser et le `ModelAdmin` qui se chargera de la personnalisation. Bien entendu, cet exemple aura un fonctionnement exactement identique à ce que vous avez vu jusqu'ici puisque la classe `ProfileAdmin`, ici, ne fait rien du tout.

Sélectionner les champs à afficher

Maintenant que vous avez écrit une classe `ProfileAdmin`, vous pouvez personnaliser l'affichage de votre liste d'objets via l'attribut `list_display`. Cette option est un tuple qui correspond à un ensemble de champs ou de méthodes du modèle sous-jacent à afficher sur votre liste, par exemple :

```
class ProfileAdmin(admin.ModelAdmin):
    list_display = ("__unicode__","commentaire")
admin.site.register(Profile, ProfileAdmin)
```

Cette fois, `__unicode__` doit être affiché car, en précisant l'option `list_display`, nous avons pris le contrôle de l'affichage et donc `__unicode__` n'est plus maintenant affiché que si nous le demandons.

Lors de vos personnalisations, vous allez rapidement être confronté à deux difficultés particulières : l'affichage des objets en relation et l'insertion de HTML dans la liste des objets.

`list_display` ne sait pas afficher des champs qui n'appartiennent pas à votre modèle. En effet, l'administration de Django est avant tout une manière d'interagir avec vos modèles. Pourtant, vous pouvez contourner ce problème de la manière la plus simple qui soit, en vous inspirant de la méthode `__unicode__`. Il s'agit d'écrire dans vos modèles une méthode qui va renvoyer un champ spécifique d'un modèle :

```
    def get_username(self):
        return self.user.username
```

puis d'utiliser cette nouvelle méthode dans `list_display`. Dans votre fichier `admin.py`, ajoutez votre nouveau champ dans `list_display` :

```
    list_display = ("__unicode__","commentaire","get_username")
```

Vous pouvez constater que votre nouveau champ s'affiche bien dans la liste des objets. Seulement, le titre de cette colonne est maintenant `get_username` ; vous auriez sans doute préféré un titre plus convivial ! Pour cela, utilisez dans vos modèles l'option `short_description` :

```
    def get_username(self):
        return self.user.username
    get_username.short_description = 'surnom'
```

Votre colonne porte maintenant le titre `surnom`.

Afficher une liste d'objets reliés dans l'interface d'administration

Dans l'exemple que nous venons de voir, il ne peut y avoir qu'un seul objet. Mais comment afficher une liste d'objets reliés ? Par exemple, imaginons que les utilisateurs écrivent des articles dont vous souhaitez afficher les titres dans l'interface d'administration.

Pour cet exemple, écrivez un modèle `Article` très simple :

```
class Article(models.Model):
    titre = models.CharField(max_length=250)
    content = models.TextField()
    publication_date = models.DateTimeField()
    author = models.ForeignKey(User)
```

Puis ajoutez à votre classe `Profile` la méthode suivante :

```
    def get_articles(self):
        return ", ".join(
            [article.titre for article in self.user.article_set.all()]
            )
    get_articles.short_description ="articles"
```

Attention, méfiez-vous : en production, `self.user.article_set.all()` est particulièrement dangereux en termes de performances.

Enfin, ajoutez simplement votre nouvelle méthode dans `list_display` :

```
    list_display = ("__unicode__","commentaire","get_username", "get_articles")
```

Remarquez qu'il n'y a pas de lien direct entre les articles et le profil. Le code de `get_article` va chercher les articles de l'utilisateur, qui récupère pour chacun le titre correspondant, puis utilise `join` pour présenter les titres en liste, séparés par des virgules. Bien entendu, cette méthode peut être affinée. Si, par exemple, vous ne voulez afficher que les cinq derniers articles, précisez-le comme suit :

```
def get_articles(self):
        return ", ".join(
            [article.titre for article in
self.user.article_set.all().order_by('-publication_date')[:5]]
            )
    get_articles.short_description ="articles"
```

Afficher du HTML dans la liste des champs

Parfois, vous aimeriez rendre l'affichage un peu plus convivial en utilisant un peu de HTML pour vos champs. Par exemple, vous pouvez afficher le champ `username` en gras si l'utilisateur est administrateur du site. Pour cela, vous avez besoin de l'option `allow_tag` :

```
    def get_username(self):
        if self.user.is_superuser:
            return "<b>%s</b>" % self.user.username
        return self.user.username
    get_username.short_description = 'username'
    get_username.allow_tags = True
```

Cet exemple est très simple, mais vous pouvez faire des choses bien plus complexes. Imaginons que vous souhaitiez créer un lien entre les titres que vous affichez et les objets `article` auxquels il font référence, afin que vos utilisateurs puissent facilement modifier ces derniers. Vous retrouvez facilement l'URL d'un objet en suivant la méthode que Django utilise pour générer les URL de l'interface d'administration.

- La page de présentation d'une application est toujours sous `/admin/<nom de l'application>`.
- La page qui présente la liste d'un modèle est sous `admin/<nom de l'application>/<nom du modèle>`.
- La page de modification d'un modèle est sous `admin/<nom de l'application>/<nom du modèle>/id de l'objet`.

Dans l'exemple présenté, l'application se nomme `journal`. Les URL sont donc les suivantes :

- page de présentation de l'application : `http://localhost:8000/admin/journal/` ;
- page qui liste tous les articles : `http://localhost:8000/admin/journal/article/` ;

- page qui présente le formulaire de modification du premier article : `http://locahost:8000/admin/journal/article/1/`.

Sachant cela, il est possible de modifier la méthode `get_articles` comme suit :

```
from django.core import urlresolvers

    def get_articles(self):
        articles = self.user.article_set.all()
        liste_titres = []
        for article in articles:
            lien = "<ahref='{0}'>{1}</a>".format(
                    urlresolvers.reverse(
                          'admin:article_article_change', args=(article.pk,)
                           ),
                    article.titre
                    )

            liste_titres.append(lien)
        return ", ".join(liste_titres)
    get_articles.short_description = "articles"
    get_articles.allow_tags = True
```

Avec ces quelques exemples, vous n'avez plus de limites à ce que vous pouvez afficher dans la liste d'objets. Prenez simplement garde à offrir à l'utilisateur les informations qui lui sont vraiment utiles. S'il y a trop d'informations, l'utilisateur risque d'être perturbé et cela ne l'aidera pas à trouver ce qu'il cherche. Sachez donc rester sobre quant à la quantité d'informations que vous lui proposez et gardez toujours à l'esprit les notions de performances. Plus vous affichez d'informations, et c'est particulièrement vrai pour les relations entre les objets, plus vous sollicitez la base de données.

Rechercher dans la liste d'objets

L'avantage de présenter une liste bien formatée est d'aider l'utilisateur à retrouver facilement un objet dans une longue liste. Django propose deux options précieuses pour rendre ce travail plus simple : un champ de recherche et des filtres.

Champs de recherche dans l'administration

Django permet d'ajouter très facilement un champ de recherche qui listera seulement les objets correspondant à certains critères. Il suffit d'ajouter un champ `search_fields` au `ModelAdmin`. Par exemple, pour faire une recherche sur le texte du champ `commentaire` de votre objet `Profile`, ajoutez simplement :

```
class ProfileAdmin(admin.ModelAdmin):
    list_display = ("__unicode__","commentaire","get_username", "get_articles")
    search_fields = ["commentaire"]
```

Et vous verrez immédiatement un champ de recherche apparaître sur votre liste `profile`.

Si vous souhaitez autoriser une recherche sur un champ lié, utilisez la syntaxe des querysets. Ainsi, pour chercher un objet `Profile` à l'aide du nom d'utilisateur auquel il est relié, vous pouvez écrire :

```
search_fields = ["user__username", "commentaire"]
```

Enfin, pour permettre de rechercher un `Profile` à l'aide des titres des articles que l'utilisateur auquel il est relié a écrits, il vous suffit d'ajouter :

```
search_fields = ["user__username", "commentaire", "user__article__titre"]
```

C'est à la fois simple et concis et cela vous donne une grande liberté pour choisir la méthode de recherche que vous allez implémenter pour votre propre application.

Les filtres dans l'administration

Outre le champ de recherche, Django vous offre également la possibilité de fournir à l'utilisateur un ensemble de filtres, qui apparaîtront à droite de la liste des objets de votre modèle et permettront à l'utilisateur de n'afficher qu'un sous-ensemble d'objets correspondant à ces critères. Un exemple de filtre en fonctionnement est visible dans l'application `User`.

Pour créer vos filtres, il suffit d'utiliser l'option `list_filters` dans `ModelAdmin`. Elle possède la même syntaxe que `search_fields`, c'est-à-dire que vous pouvez employer les relations des objets en utilisant le double _. Par exemple, pour proposer un filtre sur la date de parution des articles, vous pouvez écrire :

```
    list_filter = ["user__article__publication_date"]
```

Vous pouvez constater que Django fait un usage particulièrement intelligent des champs de type `DateTimeField` en proposant une liste de choix allant de `aujourd'hui` à `cette année`.

Si les filtres sont très pratiques pour l'utilisateur, vous devez prendre garde à n'employer que des champs qui s'y adaptent vraiment. Par exemple, si vous optez pour le nom de l'utilisateur en guise de filtre, vous aurez autant de choix que d'enregistrements ; cela risque de ne pas vous être très utile. Préférez alors comme filtres les champs de type date ou booléen.

Ordonner les résultats

Une autre fonctionnalité qui vous est offerte par Django consiste à ordonner les résultats qui vous sont présentés. Vous n'avez rien à faire quand la colonne que vous affichez correspond directement à un champ du modèle. Par exemple, cliquez sur le champ `commentaire` pour classer les résultats par ordre alphabétique ou inversement alphabétique.

Le problème se pose néanmoins quand le champ est calculé à l'aide d'une méthode de modèle, comme le champ `username`. Utilisez alors l'option `admin_order_field` sur la méthode de votre modèle pour indiquer à Django le champ de base de données qui doit être pris en compte pour l'ordonnancement :

```
    def get_username(self):
        if self.user.is_superuser:
            return "<b>%s</b>" % self.user.username
        return self.user.username
    get_username.short_description = 'username'
    get_username.allow_tags = True
    get_username.admin_order_field = "user__username"
```

En bref

Maintenant, servez-vous de l'ensemble de ces outils pour que l'utilisateur retrouve les objets qu'il souhaite consulter. En soignant ce type de détail, vous fournissez une interface simple et conviviale que les utilisateurs auront plaisir à manipuler.

Les formulaires dans l'interface d'administration

L'interface d'administration est principalement créée pour ajouter, modifier ou supprimer des objets. Il est donc très important qu'elle soit aussi simple d'utilisation que possible. Pour cela, Django vous offre un ensemble de méthodes.

Afficher les champs du modèle

Par défaut, Django va afficher tous les champs que possède le modèle. Si vous souhaitez modifier cette liste pour exclure certains champs, en ajoutant une option au `ModelAdmin` :

```
exclude = ('commentaire',)     # Exclut le champ commentaire du formulaire
d'administration.
```

Au lieu d'exclure un champ, vous pouvez au contraire préciser ceux que vous souhaitez afficher avec l'option `fields` :

```
fields = ('user',)     # N'affichera que le champs user.
```

Spécifier la valeur des champs manquants

Si vous n'affichez pas tous les champs obligatoires de vos modèles, vous devez leur fournir vous-même une valeur, en modifiant la méthode `save` soit de votre formulaire, soit de votre modèle. Dans le cas contraire, Django affichera une erreur et votre nouvel objet ne sera pas enregistré.

Personnalisez également la manière dont vos champs vont s'afficher. Par exemple, vous présenterez vos champs `user` et `commentaire` sur une seule ligne en les englobant dans un tuple spécifique :

```
fields = (('user', 'commentaire'),)
```

Pour un plus grand contrôle sur l'affichage de vos champs, n'hésitez pas à employer l'option `fieldset`. C'est une partie de formulaire qui sera affichée de manière spécifique. Sur de grands formulaires, cela conduit à une présentation plus aérée.

Pour cet exemple, il vous faut quelques champs supplémentaires sur vote modèle `Profile` :

```
class Profile(models.Model):
    user = models.OneToOneField(User)
    commentaire = models.TextField()
    couleur = models.CharField(max_length=250, null=True, blank=True)
    categorie = models.CharField(max_length=250, null=True, blank=True)
    style = models.CharField(max_length=250, null=True, blank=True)
```

Les nouveaux champs `couleur`, `categorie` et `style` sont optionnels. Vous souhaitez donc les afficher à part, via les `fieldsets` dans `ModelAdmin` :

```
fieldsets = (
        (None, {
                'fields': ('user', 'commentaire')}),
        ('Optionnel', {'fields': ('couleur', 'categorie', 'style'),
                        })
              )
```

En précisant ainsi vos champs, vous constaterez qu'ils sont maintenant séparés en deux blocs bien distincts. Vous pouvez même personnaliser encore un peu plus l'affichage de ces blocs en ajoutant à l'un d'eux une classe CSS que Django a spécialement prévue pour

vous. Les deux classes à votre disposition sont `collapse` et `wide`. Avec la première, le bloc sera « refermé » et s'ouvrira d'un clic. Avec la seconde, le bloc aura plus d'espace disponible sur la page. Vous pouvez tester ces deux classes avec le code suivant :

```
fieldsets = (
   (None, {
           'fields': ('user', 'commentaire'),
           'classes': ('wide',)
           }),
   ('Optionnel', {'fields': ('couleur', 'categorie', 'style'),
                  'classes': ('collapse',)
                  })
         )
```

Enfin, le champ `description` ajoutera une ligne de texte au-dessus de votre bloc :

```
fieldsets = (
    (None, {
           'fields': ('user', 'commentaire'),
           'classes': ('wide',),
           'description': 'Un ensemble de champs obligatoires'}),
    ('Optionnel', {'fields': ('couleur', 'categorie', 'style'),
                  'classes': ('collapse',)
                  })
         )
```

L'enregistrement des objets reliés

Vous ressentirez parfois le besoin de présenter à vos utilisateurs un formulaire d'édition d'un objet enfant sur le formulaire d'un objet parent. Prenons un cas simple : deux modèles qui partagent un lien de parentalité, par exemple un fabricant de voitures et des modèles de véhicules :

```
class Fabriquant(models.Model):
    nom = models.CharField(max_length=250)

    def __unicode__(self):
        return self.nom


class Voiture(models.Model):
    nom = models.CharField(max_length=250)
    fabriquant = models.ForeignKey(Fabriquant)

    def __unicode__(self):
        return self.nom
```

Votre fichier admin.py devrait en toute logique ressembler à ce qui suit :

```
from models import Fabriquant, Voiture
from django.contrib import admin
admin.site.register(Fabriquant)
admin.site.register(Voiture)
```

Django va alors afficher un formulaire pour la création de nouvelles voitures et un autre pour les nouvelles marques. Or, il serait beaucoup plus naturel de présenter le formulaire d'édition de nouvelles voitures directement sur la page de la marque correspondante. C'est possible à l'aide des classes InlineModelAdmin. À la manière des classes ModelAdmin, elles seront utilisées pour inclure des formulaires enfants dans un formulaire parent. Il en existe deux types, dont les différences sont surtout d'ordre esthétique : TabularInline et StackedInline. Pour inclure un formulaire d'édition des voitures dans la fiche d'un fabriquant, vous pouvez donc écrire :

```
class VoitureInline(admin.StackedInline):
    model = Voiture

class FabriquantAdmin(admin.ModelAdmin):
    inlines = [VoitureInline]

admin.site.register(Fabriquant, FabriquantAdmin)
```

Dans un premier temps, vous créez une classe VoitureInline et vous lui précisez simplement le modèle dont elle dépend. Ensuite, vous indiquez dans votre classe FabriquantAdmin que vous souhaitez inclure VoitureInline dans la liste inlines. Partant de ce principe simple, vous possédez un ensemble d'options pour contrôler l'affichage de vos ModelAdminInline. Tout d'abord, vous disposez des options déjà utilisées : fields, exclude, fieldsets... De plus, d'autres options sont spécifiques aux ModelAdminInline. Par exemple, avec l'option extra vous spécifiez le nombre de formulaires supplémentaires que Django doit afficher :

```
class VoitureInline(admin.StackedInline):
    model = Voiture
    extra = 2
```

Cet exemple affichera un total de trois formulaires (1 + 2 extra). Pour limiter le nombre de formulaires affichables, ajoutez l'option max_num :

```
class VoitureInline(admin.StackedInline):
    model = Voiture
    extra = 2
    max_num = 5
```

Cet exemple affichera trois formulaires au départ, avec un maximum de cinq.

Aller plus loin avec l'interface d'administration

Les exemples et les techniques présentées jusqu'ici conviendront pour la plupart des cas auxquels vous serez confronté. Parfois cependant, vous aurez besoin de prendre le contrôle complet de l'interface d'administration.

Les méthodes spéciales des ModelAdmin

Quand vous chercherez à prendre le contrôle complet de l'administration, vous vous retrouverez confronté à un souci majeur : vous n'avez pas accès à la requête. Par exemple, si vous voulez utiliser `request.user` pour enregistrer le nom d'un auteur d'article sans qu'il n'ait besoin de le préciser, rien ne vous permet de le faire. C'est pourquoi Django vous propose quelques méthodes qui vont justement récupérer ce type d'informations.

save_model

`save_model` est une méthode des `ModelAdmin`, qui permet de faire quelque chose de spécial sur l'objet qui va être sauvegardé avant qu'il ne soit envoyé vers la méthode `save` du modèle. Dans sa forme la plus simple, c'est-à-dire qui ne fait rien de plus que la méthode par défaut, `save_model` se présente comme suit :

```
def save_model(self, request, obj, form, change):
    obj.save()
```

Comme vous pouvez le voir, elle prend en argument :

- `self`, qui correspond à l'objet `ModelAdmin` et montre qu'il s'agit d'une méthode ;
- `request`, qui correspond à l'objet `request` que vous manipulez habituellement dans vos vues. Cet objet va vous donner accès à l'objet `user` au travers de `request.user` ;
- `obj`, qui est ici l'objet sur le point d'être sauvegardé. Si certains champs ne sont pas affichés, vous pouvez maintenant les préciser manuellement avant que votre objet ne soit renvoyé à la méthode `save` du modèle ;
- `form` est créé par Django. C'est un `ModelForm` qui peut être manipulé comme un objet `Form` classique. N'hésitez pas à relire le chapitre 13 « Dialogue avec l'utilisateur : les formulaires » ;
- `change` est un booléen. Il s'agit d'une réponse donnée par Django : si l'objet est initialement créé, la réponse sera alors `False` et si l'objet est seulement modifié, la réponse sera alors `True`.

Enfin, après avoir utilisé cette méthode, il est très important que vous appeliez `obj.save()` pour que l'objet que vous manipulez soit renvoyé à la véritable méthode `save` du modèle et que le traitement du formulaire suive son cours.

delete_model

Cette méthode est appelée lorsqu'un utilisateur, au travers de l'interface d'administration, choisit de supprimer un objet. Sa syntaxe est plus simple :

```
def delete_model(self, request, obj):
   obj.delete()
```

De la même manière que `save_model`, `delete_model` vous donne accès à la `request` et à l'objet qui va être supprimé, `obj`. Vous pouvez donc manipuler ces deux objets avant d'appeler `delete` sur l'objet.

Modifier les formulaires de l'administration

Django crée pour vous les formulaires que vous employez dans l'interface d'administration. Il s'agit de `ModelForm`, c'est-à-dire de modèles dont les champs sont automatiquement créés pour correspondre aux champs du modèle dont ils découlent.

L'une des manières les plus efficaces de prendre le contrôle de l'affichage des formulaires consiste à préciser à Django quels formulaires utiliser. Lorsque c'est fait, les limites de l'administration ne sont plus que de l'histoire ancienne. En effet, vous avez à votre disposition toutes les options et toutes les méthodes des formulaires dans l'administration. Vous pouvez, par exemple, créer de nouveaux widgets pour afficher de nouveaux champs, créer des validateurs spéciaux… Pour indiquer à Django d'utiliser un formulaire spécifique, il suffit de l'indiquer dans une classe `ModelAdmin` :

```
class ProfileAdmin(admin.ModelAdmin):
    form = ProfileAdminForm
```

Votre formulaire, dans sa forme la plus simple s'écrit de la façon suivante :

```
class ProfileAdminForm(forms.ModelForm):
    class Meta:
        model = Profile
```

En bref

Le module `admin` de Django est certainement l'application la plus vaste et la plus complexe mais aussi la plus puissante que peut vous offrir Django. Vous découvrirez sans doute, au fil de vos développements, de nouvelles astuces pour en faire plus avec

l'administration. Il est possible de construire des sites Internet entièrement au travers de cette interface ; ses possibilités sont vraiment impressionnantes. En réalité, beaucoup de bonnes idées, comme les `ModelForm`, ont été conçues au départ pour ce module et se sont petit à petit étendues au reste du framework.

Cependant, ne perdez pas de vue qu'il s'agit finalement d'une application tout ce qu'il y a de plus banal. Ainsi, même si nous ne l'avons pas traité ici, il est tout à fait possible d'effectuer du surclassage de templates. N'hésitez donc pas à le faire afin de personnaliser encore plus votre interface.

Au fil des versions de Django, cette application vous offre toujours plus de contrôle et de simplicité. En effet, ce qui demandait hier beaucoup de lignes de code est aujourd'hui bien plus simple à réaliser.

Les autres modules contribs

Trop souvent, les contribs de Django que l'on utilise se résument à `django.contrib.auth` et `django.contrib.admin`. Pourtant, il en existe bien d'autres, certes moins utiles mais tout aussi plaisants à employer.

Les commentaires

Le principe de Django est d'être livré avec des piles. Sous ce vocable se cache la volonté de permettre aux développeurs d'applications web de créer un site Internet complet avec Django sans nécessairement avoir besoin de réécrire sans cesse les même applications.

La logique de l'application `comments` se veut aussi simple, souple et extensible que possible. Son but est d'attacher des commentaires sur un modèle, quel qu'il soit : on pense bien entendu à un article de blog, mais vous pouvez tout aussi bien attacher un commentaire à une photo, un utilisateur…

L'architecture des commentaires de Django est écrite de telle façon que vous n'aurez jamais besoin de modifier l'objet ou le modèle auquel est attaché un commentaire ; le commentaire et l'objet ne sont pas correlés. Cela signifie que vous pouvez brancher le système de commentaires sur une application qui n'a pas du tout été prévue en ce sens.

Le système de commentaires se veut également très simple à mettre en place. Concrètement, vous n'aurez qu'à ajouter une ligne dans votre `settings.py` et une URL dans le fichier racine `urls.py`, puis à créer les tables dans la base de données ; vous pourrez alors immédiatement vous servir de l'application.

Activer les commentaires pour votre projet

Lorsque vous avez besoin d'attacher des commentaires à un modèle, ajoutez l'application `django.contrib.comments` à vos `INSTALLED_APPS`, dans votre `settings.py`. Branchez les URL de l'application dans `urls.py` :

```
url(r'^comments/', include('django.contrib.comments.urls')),
```

Enfin, vous devez créer les tables dans la base de données dont l'application a besoin :

```
$ python manage.py runserver
```

Rien de plus n'est nécessaire pour rendre fonctionnelle l'application `comments`.

Utilisation des commentaires dans les templates

L'emploi du système de commentaires que vous propose Django va se faire principalement dans les templates. Vous pourrez simplement afficher une liste de commentaires ainsi qu'un formulaire pour un objet. Lorsque vous souhaitez employer l'une ou l'autre de ces fonctionnalités, vous devez en informer le système de templates via le template tag `{% load %}` :

```
{% load comments %}
```

Afficher la liste des commentaires

Une fois que vous avez chargé les template tags spécifiques de l'application `comments`, vous pouvez afficher la liste des commentaires d'un objet en écrivant par exemple :

```
{% for article in articles %}
{% render_comment_list for article %}
{% endfor %}
```

Si vous le souhaitez, il est possible d'afficher le nombre de commentaires pour chaque article avec la syntaxe suivante :

```
{% get_comment_count for article as count %}
{% count %}
```

Afficher un formulaire d'ajout de commentaire

De la même manière, un formulaire de commentaire peut se présenter de façon très simple. Il suffit cette fois d'utiliser :

```
{% render_comment_form for article %}
```

Django affiche un formulaire de commentaire et s'occupe à la fois de valider les données de l'utilisateur, mais également de vous offrir quelques protections. Tout d'abord, le template est livré avec une protection contre les CSRF, comme tous les formulaires que vous créez avec Django, mais il est également livré avec un champ `honeypot` (« pot de miel ») qui est un piège pour lutter contre le spam. Il s'agit d'un champ caché à l'utilisateur normal et qui ne doit pas être rempli pour que le formulaire puisse être enregistré. De cette façon, un robot à spam, qui va tenter de remplir tous les champs y compris celui qui ne doit pas l'être, ne pourra pas enregistrer « son » formulaire.

Prendre le contrôle des commentaires Django

Maintenant que vous connaissez bien Django, vous y êtes habitué : si la mise en place est très simple, vous avez tout de même la possibilité de modifier les comportements par défaut au fur et à mesure que votre besoin de contrôle devient plus grand. Votre prise de contrôle des commentaires se fera directement au niveau du template.

Modifier l'affichage des commentaires

Pour modifier les commentaires, vous devez prendre le contrôle de l'affichage, ce que ne vous offre pas la syntaxe `render`. Affichez votre liste de commentaires avec la syntaxe suivante :

```
{% get_comment_list for article as comments %}
{% for comment in comments %}
<ici votre code>
{% endfor %}
```

Vous pouvez ainsi itérer comme il vous plaît sur les commentaires de votre article, y insérer des classes, représenter les commentaires sous forme de liste, de table… De la même manière, pour modifier l'affichage du formulaire de commentaire, il est possible d'employer une syntaxe très similaire :

```
{% get_comment_form for article as form %}
```

Vous l'aurez compris, vous avez maintenant une variable `form` dans votre template, qui contient le formulaire de commentaire, qui se comporte exactement de la même manière que les formulaires classiques. Ainsi, vous pouvez utiliser `form.as_ul`, `form.as_p`…

Il vous reste néanmoins un problème à régler pour rendre fonctionnel votre formulaire. Contrairement à un formulaire que vous avez créé vous-même, vous ne possédez pas l'URL qui pointera vers sa validation. En effet, c'est directement Django, au travers de l'application de commentaires, qui se charge de cette validation. C'est pour cela qu'il vous fournit `{% comment_form_target %}`. Ce template tag pointe directement vers la validation du formulaire de commentaire.

Une dernière chose concerne la redirection après validation. Par défaut, Django renvoie l'utilisateur sur une page de remerciements directement intégrée dans l'application `comments`. Si vous souhaitez modifier ce comportement et le rediriger sur une page spécifique, utilisez le champ caché `next` :

```
<input type="hidden" name="next" value="url_de_redirection" />
```

En bref

L'application `django.contrib.comments` ajoute très simplement des commentaires à votre application. Elle est également très intéressante du point de vue de l'architecture générale : en quelques template tags, elle vous offre toutes les fonctionnalités dont vous avez besoin. C'est un très bon exemple de ce que doit être une application réutilisable. N'hésitez donc pas à lire son code et à vous en inspirer pour la création de vos propres applications réutilisables.

Les content-types

Lorsque vous souhaiterez créer une application réutilisable, vous verrez souvent comme les content-types peuvent vous être d'un grand secours. En effet, ce système connecte entre elles des ressources qui ne se connaissent pas. Ainsi, vous êtes à même de créer, par exemple, une application qui a vocation à enrichir un modèle sans le connaître par avance. Mieux, votre application peut se connecter à plusieurs modèles différents sans avoir conscience de leur existence : c'est, par exemple, ce que vous propose l'application `comments`. Lorsque vous utilisez cette application, Django enregistre pour tous vos objet une sorte de « carte d'identité » qui va permettre de le retrouver facilement lorsqu'il s'agira de le lier avec un objet d'un autre type.

Une application de tag utilisant les content-types

Pour bien comprendre le fonctionnement des content-types, nous allons créer une application de tag très simple. L'idée est de permettre de taguer n'importe quel objet d'une application.

Installation

L'installation des content-types est assez rapide. Il est d'ailleurs fort possible que cette application soit déjà fonctionnelle dans votre projet, car elle est habituellement activée par défaut. Néanmoins, si vous avez modifié votre `settings.py`, vous devrez effectuer les modifications suivantes pour activer les content-types :

- ajouter `'django.contrib.contenttypes'` à la liste de vos `INSTALLED_APPS` ;
- lancer la commande `python manage.py syncdb` pour créer les tables nécessaires.

Une fois les contenttypes activés, créez votre application de tag :

```
python manage.py startapp tagging
```

Vous pouvez ensuite éditer les modèles de votre nouvelle application :

```
# tagging/models.py
from django.db import models
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes import generic

class Tag(models.Model) :
    nom = models.CharField(max_length=250)
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericForeignKey(
                        'content_type',
                        'object_id')
```

Dans un premier temps, vous importez tout ce dont vous allez avoir besoin pour que votre application soit fonctionnelle, c'est-à-dire ContentType, qui servira pour lier votre modèle à un objet de ce type, et generic qui va permettre de créer une relation « générique ».

Le modèle ContentType possède la structure suivante :

- app_name : le nom de l'application d'un objet ;
- model : le nom du modèle de l'objet ;
- name : le nom du modèle lisible par des humains. Il est créé en utilisant le paramètre verbose_name du modèle.

En faisant des requêtes sur le modèle ContentType, vous pourrez retrouver les modèles de toutes les applications de votre projet. C'est particulièrement utile, mais cela ne vous servira pas pour retrouver un objet spécifique. C'est pourquoi vous aurez également besoin de l'identifiant unique d'un objet :

```
object_id = models.PositiveIntegerField()
```

Dans une relation habituelle, vous ne pourriez lier les objets de type Tag que vers un modèle unique, mais avec l'utilisation de generic.GenericForeignKey, vous serez en mesure de les lier à n'importe quel objet de n'importe quel modèle. En effet, vous indiquez à GenericForeignKey à la fois un modèle, avec ContentType, et un identifiant unique, avec object_id.

Les requêtes génériques

Maintenant que votre modèle est écrit, commencez à l'utiliser. Par exemple, ajoutez un tag à un objet `User` :

```
>>> from django.contrib.auth.models import User
>>> from tagging.models import Tag
>>> user = User.objects.get(username='yohann')
>>> tag = Tag(content_object=user, name='auteur')
>>> tag.save()
```

Cette syntaxe vous permet donc de lier les tags à des modèles que vous ne connaissez pas au départ. Nulle part dans votre code vous n'avez fait appel au modèle `User` et, bien entendu, ce dernier n'a absolument pas conscience de votre application de tag.

Ce qu'il faut maintenant, c'est retrouver tous les objets liés à un tag et tous les tags liés à un objet. Comme les relations génériques ne sont pas « normales », vous devez fournir un peu de travail pour obtenir ce résultat.

Retrouver tous les tags d'un objet

Vous devez tout d'abord charger l'objet :

```
user = User.objects.get(username='yohann')
```

Ensuite, il faut retrouver l'objet `ContentType` qui lui correspond. Pour cela, Django vous offre la méthode `get_for_model` du manager de `ContentType` :

```
>>> contenttype = ContentType.objects.get_for_model(user)
```

Maintenant, vous avez tous les éléments pour retrouver les tags placés sur l'objet `User` :

```
>>> tags = Tag.objects.get(content_type__pk= contenttype.pk, object_id=user.id)
<Tag:auteur>
```

Retrouver tous les objets d'un tag

Si vous connaissez le tag (`auteur` dans l'exemple qui suit) et souhaitez retrouver les objets correspondants, c'est bien plus simple :

```
tag = Tag.objects.filter(name='auteur')
```

En bref

L'application `ContentType` est très puissante mais quelque peu perturbante au premier abord. Avec un peu d'habitude, vous ne vous perdrez plus dans ces relations très génériques.

Relisez le code des commentaires de Django, qui vous donneront une vision d'ensemble plus claire sur cette partie du framework.

N'hésitez pas pour autant à vous servir de `ContentType`, qui vous offre la possibilité de mettre en place des applications vraiment très génériques.

L'application flatpages : gérer les parties statiques de votre site

Lorsque vous créez une application web, vos pages sont généralement dynamiques. Elles utilisent des informations issues de bases de données, des entrées des utilisateurs, d'API distantes, etc. Pourtant, certaines parties de votre site ne correspondent pas à cette description. Prenons, par exemple, la page « à propos » : c'est tout simplement du HTML statique. Pour créer cette page, vous pouvez bien entendu écrire un template, ajouter une URL dans votre fichier `urls.py` et la mettre à jour directement sur votre serveur. Avouez que ce n'est pas très confortable.

L'application `flatpages` sert à créer ce type de page directement dans l'`admin` de Django. Le HTML que vous y écrirez sera alors sauvegardé directement en base de données, vous permettant de le modifier simplement.

Installation

L'installation de `flatpages` nécessite que vous ayez mis en place l'application `django.contrib.sites`. Vous devez également ajouter `django.contrib.flatpages.middleware.FlatpageFallbackMiddleware` à votre liste de `MIDDLEWARE_CLASSES` dans `settings.py`. Puis ajoutez l'application elle-même dans la liste des `INSTALLED_APPS`, toujours dans `settings.py` :

```
'django.contrib.flatpages'
```

Pour mettre à jour votre base de données, utilisez la commande habituelle :

```
$ python manage.py syncdb
```

Fonctionnement

Comme vous utilisez le middleware `flatpages`, vous n'avez pas besoin d'ajouter quoi que ce soit aux URL. En réalité, dès que Django s'apprête à afficher une page 404, il

va chercher une flatpage qui aurait l'URL demandée par l'utilisateur. Dans le cas où il trouve une telle page, il l'affiche au lieu de la page 404.

Vous pouvez donc à présent écrire vos flatpages. Par défaut, vous pourrez choisir une URL, un titre et y écrire le contenu de votre page. Si leur fonctionnement est très simple et assez peu sujet à une configuration particulière, il est tout de même possible de définir quelques éléments :

- employer des commentaires sur chacune d'elles. Il faut pour cela que l'application `comments` soit activée pour votre projet en cours ;
- limiter l'accès à une flatpage aux seuls utilisateurs enregistrés ;
- définir un template spécifique pour une flatpage ou plusieurs ;
- sélectionner plusieurs objets `site` qui utiliseront une flatpage commune grâce à l'application `sites` de Django.

Le framework de message

Imaginons la situation suivante : sur votre application, vous proposez un formulaire de contact. Lorsque l'utilisateur le soumet, vous souhaitez le remercier et lui indiquer que vous avez bien pris en compte son message. Ensuite, vous le redirigez vers la page d'accueil. Il vous faut donc un système avertissant l'utilisateur sans employer une page intermédiaire. C'est très exactement ce que permet `django.contrib.message`.

Installation

Les messages de Django n'ont pas besoin de table en base de données pour fonctionner. Ils utilisent uniquement un middleware et un context processor qui vont stocker les messages dans le `Context`.

Dans une installation classique de Django, les messages sont déjà disponibles. Vous n'avez donc rien à faire de particulier pour les utiliser. Cependant, dans le cas où vous auriez modifié `settings.py`, vous pouvez les réactiver en ajoutant :

- `'django.contrib.messages'` à la liste des `INSTALLED_APPS` de `settings.py` ;
- `'django.contrib.sessions.middleware.SessionMiddleware'` au système employant la session pour enregistrer les messages, les `MIDDLEWARE_CLASSES` ;
- `'django.contrib.messages.middleware.MessageMiddleware'` à vos `MIDDLEWARE_CLASSES`.

Utilisation

Le système de message emploie un concept de « niveaux » qui vous donne un contrôle fin sur la manière dont les messages vont s'afficher. Par défaut, Django implémente cinq niveaux de message : `debug`, `info`, `success`, `warning` et `error`.

Les messages dans les vues

Pour créer un nouveau message dans une vue, employez la syntaxe suivante :

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'Merci de nous avoir contactés.
Nous vous répondrons dès que possible.')
```

Les paramètres de `message.add_message` sont :

- l'objet `request` ;
- le niveau de message que vous souhaitez utiliser ;
- le message que vous souhaitez afficher.

Les messages dans les templates

Avec `message.add_message`, Django ajoute au contexte une liste de messages que vous pouvez afficher dans un template. Pour cela, vous pouvez donc simplement employer un fragment HTML de ce type :

```
{% if messages %}
<ul>
    {% for message in messages %}
    <li{% if message.tags %} class="{{ message.tags }}"{% endif %}>{{ message
}}</li>
    {% endfor %}
</ul>
{% endif %}
```

Ce peut être une bonne idée d'ajouter ce fragment à vos `includes` par défaut, ce qui vous permettra d'utiliser les messages à tout endroit de votre application.

Comme l'application `message` utilise un système de session, les messages peuvent être envoyés sans que vos utilisateurs aient besoin de s'authentifier sur votre site. C'est en somme une contribution à Django qui est à la fois très simple d'utilisation et extrêmement pratique.

L'application sitemaps : produire une carte de votre site

Pour que votre toute nouvelle application web ait du succès, il faut que son référencement sur les moteurs de recherche soit performant. C'est pour vous y aider que les développeurs de Django ont créé l'application `sitemaps`, laquelle produit une carte de votre site qui en présente les ressources sous forme hiérarchique. En 2005, Google a proposé le protocole *sitemap* édictant des règles pour présenter ce type de ressources sous le format XML et à destination exclusive des moteurs de recherche. Ce protocole est employé aujourd'hui par tous les grands acteurs du marché : Yahoo!, Bing, Exalead et, bien entendu, Google.

Installation

L'application `sitemaps` de Django ne nécessite pas de table en base de données. Son installation en est donc simplifiée. Il vous faut ajouter la ligne suivante à la liste de vos `INSTALLED_APPS` :

```
'django.contrib.sitemaps'
```

Vous devez également ajouter une URL qui pointera vers le fichier autogénéré par Django, par exemple :

```
(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps':
sitemaps})
```

Cela indique à Django de créer un fichier `sitemap.xml` quand un utilisateur ou un robot d'indexation se rend à l'URL `http://votresite/sitemap.xml`.

Cette URL prend en argument un dictionnaire ne contenant qu'une seule clé `sitemaps` qui pointe vers la variable `sitemaps` pour le moment inexistante. Il va falloir créer cette variable comme une instance de la classe `Sitemap`.

La classe Sitemap

La classe `Sitemap` vous est fournie par Django. Il vous suffit donc de la surclasser pour qu'elle corresponde à vos besoins :

```
from django.contrib.sitemaps import Sitemap
```

- `items` va dresser une liste d'objets à inclure dans votre sitemap ;

Vous devrez définir deux méthodes sur votre classe `Sitemap` :

- `last_mod` va indiquer le champ de votre modèle à utiliser pour préciser la date de dernière modification.

Votre classe `Sitemap` peut donc s'écrire comme suit :

```
from monapplication.models import MonModel

class MonSiteSitemap(Sitemap):

    def items(self):
        return MonModel.objects.all()

    def lastmod(self, obj):
        return obj.modification_date
```

Vous pouvez également préciser la variable `changefreq` qui donne aux moteurs de recherche une idée de la fréquence à laquelle ils doivent consulter de nouveau votre sitemap pour rafraîchir leurs index. Les choix possibles sont : `always` (toujours), `hourly` (toutes les heures), `daily` (une fois par jour), `weekly` (une fois par semaine), `monthly` (une fois par mois), `yearly` (une fois par an) et `never` (jamais).

La variable `priority` sert à évaluer l'importance d'une page de votre site par rapport aux autres. Sa valeur de base est de `0.5` et elle peut aller de `0.0` à `1.0`. Ces variables optionnelles peuvent être indiquées au début de votre classe, par exemple :

```
from monapplication.models import MonModel

class MonSiteSitemap(Sitemap):
    priority = 0.8
    changefreq = 'monthly'
    def items(self):
        return MonModel.objects.all()

    def lastmod(self, obj):
        return obj.modification_date
```

Les raccourcis pour la création de sitemaps

Django vous offre également la possibilité de gérer les cas fréquents de création de sitemaps en utilisant une syntaxe simplifiée et plus rapide avec `GenericSiteMap`.

```
from django.contrib.sitemaps import GenericSitemap
```

Définissez simplement un dictionnaire contenant les informations dont votre sitemap a besoin :

```
info_dict = {
    'queryset': MonModel.objects.all(),
    'date_field': 'date_de_creation',
}
```

Utilisez ensuite ce dictionnaire pour créer directement un sitemap :

```
sitemaps = {
    'articles': GenericSitemap(info_dict, priority=0.6),
}
```

Voici enfin l'URL :

```
(r'^sitemap\.xml$', 'django.contrib.sitemaps.views.sitemap', {'sitemaps':
sitemaps})
```

En bref

La création de sitemaps n'a jamais été plus facile qu'avec Django. Que vous utilisiez la première ou la seconde syntaxe, suivant vos besoins, vous n'aurez aucun mal à produire un sitemap de qualité en quelques minutes.

Le framework de site : gérer plusieurs sites à la fois

L'une des fonctionnalités les moins connues de Django est sa capacité à gérer plusieurs sites à la fois. C'est pourtant extrêmement pratique à bien des égards.

Imaginons, par exemple, que vous ayez développé une nouvelle application. Vous souhaitez sans doute proposer un blog qui donnera à vos utilisateurs des nouvelles sur les évolutions de vos produits. Plutôt que de créer deux projets différents pour les deux produits, vous devriez n'en faire qu'un mais servant les deux espaces, l'application et le blog, sur des URL différentes.

Même si vous ne souhaitez pas vous servir de cette fonctionnalité, quelques-uns des modules que nous avons traités dans cet ouvrage emploient le framework de site de Django. C'est donc toujours une bonne idée que de voir comment s'en servir efficacement.

Installation

Le framework de site est activé par défaut sur les instances de Django. Le système d'authentification, les commentaires, les flatpages et d'autres applications l'emploient. Dans le cas où vous l'auriez désactivé, réactivez-le en ajoutant la ligne suivante à vos `INSTALLED_APPS` :

```
'django.contrib.sites',
```

Puis créez la table utilisée par le framework de site :

```
$ python manage.py syncdb
```

Cette table est très simple. Elle contient un champ `domain` qui correspond au domaine sur lequel votre site est hébergé et un champ `name` qui se rapporte au nom de votre site tel qu'il sera affiché aux internautes.

Utilisation

Le framework de site va vous permettre, pour deux sites différents, d'utiliser les mêmes applications mais d'effectuer un traitement différent sur certaines actions suivant le site sur lequel l'utilisateur se trouve.

Lier un contenu à un site

Avec le framework `site`, vous pouvez rendre certaines ressources disponibles sur plusieurs sites. Commencez par éditer votre contenu pour qu'il soit possible de préciser les sites qui pourront l'afficher. Par exemple :

```
from django.db import models
from django.contrib.sites.models import Site

class MonModel(models.Model) :
    <autres champs>
    sites = models.ManyToManyField(Site)
```

Il faut maintenant que le code qui récupère votre contenu, une vue le plus souvent, puisse filtrer les contenus à afficher. Pour cela, la fonction `get_current_site` récupère l'objet `site` de votre instance et ainsi vérifie quel contenu doit être publié ou non.

```
from django.contrib.sites.models import get_current_site

def ma_vue(request) :
    objets =
MonModel.objects.filter(sites__id__exact=get_current_site(request).id)
```

Bien entendu, si vous souhaitez lier un contenu à un seul site au lieu de plusieurs, employez une foreign key au lieu d'une many-to-many :

```
class MonModel(models.Model) :
    <autres champs>
    sites = models.ForeignKey(Site)
```

Obtenir le site courant sans la requête

Si les exemples que nous venons de voir sont assez simples, il arrive parfois que vous n'ayez pas accès à la requête. C'est le cas, par exemple, dans une classe formulaire. Vous ne pouvez donc pas utiliser :

```
get_current_site(request)
```

Toutefois, Django est tout de même capable de récupérer les informations dont vous avez besoin avec la méthode `get_current` du manager du modèle `Site` :

```
from django.contrib.sites.models import Site
site = Site.objects.get_current()
```

La mise en place de contenu pour plusieurs sites ne pose pas vraiment de problèmes avec Django. Le framework `sites` gère en effet ce type de cas avec simplicité.

Le framework de syndication : créer un flux RSS

Il est courant de proposer aux internautes de consulter votre site au travers de flux RSS. Ces derniers abonnent les internautes qui le souhaitent aux derniers changements de votre site. Ainsi, quand vous écrivez un nouvel article par exemple, l'utilisateur reçoit une notification dans son lecteur de flux RSS. Il lui est alors possible de lire l'article depuis son lecteur, ou seulement une introduction suivant la manière dont vous souhaitez qu'il profite de votre contenu.

Avec Django, vous avez à votre disposition un ensemble d'outils pour créer automatiquement un flux RSS. Le mode de fonctionnement est assez proche de celui de `sitemaps` : publier automatiquement vos contenus sous un format différent.

Installation

L'installation du système de syndication est très simple : ajoutez `django.contrib.syndication` à vos `INSTALLED_APPS`.

Utilisation

Le principe de base de la gestion des flux RSS dans Django consiste à écrire une classe par type de contenu et à brancher une URL à cette liste. Django s'occupe alors du reste.

La classe que vous allez écrire hérite de `django.contrib.syndication.views.Feed`. Pour qu'elle soit valide, elle doit définir :

- un champ `title` qui correspondra au titre du contenu ;
- un champ `link` qui devra être une URL pointant vers la ressource originelle sur votre site Internet ;
- un champ `description` qui contiendra une description de votre contenu.

Enfin, cette classe doit contenir des méthodes qui vont définir ces même champs pour les entrées de votre flux RSS. Ce dernier va, la plupart du temps, correspondre à un modèle ; vous pouvez donc utiliser facilement les méthodes que vous avez déjà définies dans celui-ci. Par exemple, vous pouvez créer le champ `title` à partir de la méthode `__unicode__` et le champ `link` à partir de la méthode `get_absolute_url`. Pour clarifier tout cela, prenons l'exemple d'un agenda :

```
from agenda.models import Event
from django.contrib.syndication.views import Feed
```

```
class DerniersEvenements(Feed):
    title = "Les derniers événements de l'agenda"
    link = "/agenda/"
    description = "Les derniers événements publiés dans l'agenda"


    def items(self):
        return Event.objects.order_by('-date')[:5]

    def item_title(self, item):
        return item.titre

    def item_description(self, item):
        return item.description
```

Votre classe `DerniersEvenements` propose bien les champs `title`, `link` et `description`. Vous spécifiez les objets à retourner dans la méthode `items`. En précisant `Event.objects.order_by('-date')[:5]`, vous retournez les cinq derniers événements. Le champ `link` de vos items est automatiquement déduit de la méthode `get_absolute_url` de votre modèle. Si cette méthode n'existe pas, vous devez indiquer, dans votre classe de flux, une méthode qui retrouve cette URL, par exemple :

```
def item_link(self, item):
    return '/agenda/%s/%i' % (item.day, item.pk)
```

Il ne vous reste plus qu'à créer une URL qui va publier ce flux RSS :

```
from django.conf.urls import patterns, url, include
from agenda.feeds import DerniersEvenements

urlpatterns = patterns('',
    # ...
    (r'^flux/event/$', DerniersEvenements()),
```

Les utilisateurs de votre site pourront alors s'abonner à votre nouveau flux RSS en se rendant à `http://votresite/flux/event`.

Aller plus loin avec les flux RSS

Nous ne vous avons montré qu'un aperçu de l'utilisation des flux RSS. Elle couvre déjà la majorité des cas d'utilisation, mais il est possible de configurer plus finement vos flux. C'est assez simple mais demande d'avoir une connaissance approfondie des normes qui régissent la publication de flux. Par exemple, en lisant la norme actuellement en vigueur disponible en français (http://www.scriptol.fr/rss/RSS-2.0.html), vous constatez qu'il est possible d'inclure dans votre flux RSS la langue dans laquelle est

rédigée votre ressource. Si vous souhaitez ajouter cet élément, vous devez simplement l'indiquer à Django comme suit :

```
class DerniersEvenements(Feed):
    title = "Les derniers événements de l'agenda"
    link = "/agenda/"
    description = "Les derniers événements publiés dans l'agenda"
    language = 'fr-FR'
```

Ajoutez de la même manière des champs sur vos éléments qui sont présents dans la norme. Par exemple, vous pouvez donner un champ `author`, car c'est un élément présent dans la norme. Pour cela, vous n'avez qu'à créer la méthode `author_name` :

```
def author_name(self, item):
    return item.author.name
```

En bref

La création de flux RSS se veut la plus simple qui soit avec Django. Cette application vous y aidera grandement. À chaque fois que vous proposez un site Internet offrant du contenu (blog, journal, agenda…), il est pertinent de penser aux autres manières de publier du contenu que le site web. En prenant exemple sur le moteur de syndication de Django, vous pouvez imaginer des classes qui proposeront des syndications basées sur les réseaux sociaux par exemple, comme Facebook ou Twitter.

Conclusion

Bien que les contribs ne sont pas réellement nécessaires au fonctionnement d'un projet Django, ces applications possèdent un ensemble de fonctionnalités que l'on retrouve très souvent dans un projet web. Souvent écrites de façon générique et réutilisable, elles vous permettront de gagner énormément de temps de développement. N'hésitez donc pas à les utiliser et à lire leur code source, car les contribs sont de très bons exemples de ce qu'il faut faire pour rendre une application générique.

16

Django avancé

Grâce à une bonne connaissance de Python et de Django, vous pourrez solutionner bon nombre de problèmes courants. Maîtrise de l'interface d'administration, surclassement des templates et même création des applications génériques en jouant avec les rouages de Python au travers des générateurs de classes, etc., nous allons ici vous expliquer comment en faire plus avec Django.

Percer les secrets de l'interface d'administration

L'interface d'administration de Django est un formidable outil pour qui veut proposer rapidement une interface permettant de manipuler les données du site. Ces possibilités de modifications sont très étendues. Pourtant, du fait de sa nature autocréée, l'interface d'administration ne le peut pas tout, du moins pas sans votre aide.

Nous allons donc ici découvrir quelques astuces et bonnes pratiques pour l'aider à faire exactement ce que vous voulez. À travers quelques exemples, vous constaterez qu'avec une bonne connaissance de Django et, surtout, de Python, peu de choses sont impossibles.

Les templates de l'administration

On l'oublie bien souvent, tant elle prend une place importante dans bien des projets Django, mais `admin` n'est qu'une application comme les autres. Ainsi, les mêmes règles de surclassement des templates s'y appliquent.

Révision des règles de surclassement de templates

Avant de rentrer dans les finesses et les astuces de surclassement de templates, voyons comment ce système fonctionne normalement. Habituellement, chaque application est dotée de son dossier `templates`. Par exemple, si l'application `journal` réside dans le projet `agenda`, vous trouverez un dossier `templates` au niveau suivant :

```
agenda/journal/templates
```

Ce dossier va contenir l'ensemble des templates nécessaires à l'application, en reprenant les règles de nommage `<nom d'application>/templates`. Par exemple, le template de la page d'accueil de l'application `journal` se trouvera dans :

```
/agenda/journal/templates/journal/accueil.html
```

Le nom de l'application, ici `journal` est répété deux fois dans le chemin. Cela paraît redondant, mais c'est utile surtout pour modifier les templates d'une application dans une autre. Par exemple, l'application `agenda` peut redéfinir une page d'accueil en modifiant celle de l'application `email` :

```
/agenda/journal/templates/email/accueil.html
```

Toutefois, cela pose un problème de taille. Si vous redéfinissez de cette manière les templates, comment savoir lequel de `/agenda/email/templates/email/accueil.html` et de `/agenda/journal/templates/email/accueil.html` va être appelé quand votre vue demandera le template `email/accueil.html` ?

Revenons sur la façon dont les templates sont sélectionnés par Django. Lorsque vous demandez `email/accueil.html`, Django cherche un template avec un nom de fichier correspondant jusqu'à trouver un candidat valide et le rendre. En premier lieu, Django cherche dans le dossier `templates` à la racine de votre projet, puis dans chaque dossier `templates` à la racine de vos applications dans l'ordre de la liste des `INSTALLED_APPS`. Dans le cas présent, si votre liste `INSTALLED_APPS` est de la forme suivante :

```
INSTALLED_APPS = ['email', 'journal']
```

c'est le template `/agenda/email/templates/email/accueil.html` qui sera rendu. Dans le cas contraire, c'est-à-dire :

```
INSTALLED_APPS =['journal', 'email']
```

c'est le template `agenda/journal/templates/email/accueil.html` qui sera sélectionné.

Une application thème

À première vue, vous vous dites certainement que jouer avec l'ordre des INSTALLED_APPS pour sélectionner l'ordre des templates est une bien mauvaise idée. Avoir les templates d'une application dans une autre est en effet particulièrement néfaste pour le principe du DRY, car vous perdez le second terme du principe DRY : « La ressource doit être à un endroit clairement identifiable. »

Pourtant, il est un cas où cela peut être très pratique, et même recommandé ; il concerne l'application theme. Continuez à écrire les templates dans les applications elles-mêmes, car vous maintiendrez la réutilisabilité de ces dernières. Seulement, lorsque vous souhaitez appliquer des modifications à un template de base, plutôt que de définir un dossier templates à la racine de votre projet, créez une application theme dans laquelle vous définirez vos templates modifiés. Ainsi, cette application est portable et sera aisément installée ou désinstallée. Bien entendu, cette application theme devra être listée en premier dans vos INSTALLED_APPS afin de prendre le pas sur les autres templates définis par applications. Vous pouvez également créer plusieurs applications theme pour un même projet et en changer en fonction de vos besoins.

Surclassement des templates d'administration

Les templates d'administration les plus facilement modifiables sont les suivants :

- app_index.html
- change_form.html
- change_list.html
- delete_confirmation.html
- object_history.html

Ils existent chacun en deux versions :

- admin/app_index.html
- admin/<application>/app_index.html

Cela signifie que vous pouvez modifier les templates de l'admin pour l'ensemble de votre site ou au cas par cas, application par application. De plus, l'écriture modulaire des templates de l'admin de Django vous permettent également de ne surclasser qu'une petite partie du template et non l'ensemble de celui-ci (qui peut être un peu complexe parfois.)

Surclasser change_list_result

Les règles de surclassage des templates répondent à la très grande majorité des cas. Pourtant, il est certaines parties de l'admin que vous ne pouvez pas modifier aussi aisément. L'une d'elles est change_list_results, qui se trouve dans le template

change_list.html. Ce bloc retourne la liste des résultats, le tableau de change_list.html. Voici comment il se présente :

```
{% block result_list %} {% if action_form and actions_on_top and
cl.full_result_count %}
{% admin_actions %} {% endif %} {% result_list cl %} {% if action_form and
actions_on_bottom and cl.full_result_count %}
{% admin_actions %} {% endif %}
{% endblock %}
```

change_list_result est un template tag qui prend cl en argument et renvoie un tableau HTML. Vous devez donc créer un nouveau template tag qui aura le même comportement mais qui va renvoyer un nouveau template pour afficher ses résultats :

```
def result_list_custom(cl):
    """ Displays the headers and data list together """
    return {'cl': cl,
            'result_hidden_fields' :list(
                  result_hidden_fields(cl)),
            'result_headers': list(result_headers(cl)),
            'results': list(results(cl))}
    template = "admin/options/change_list_results_custom.html"
result_list_custom = register.inclusion_tag(template)(result_list)
```

Une fois cela fait, vous pouvez modifier le template change_list.html :

```
{% block result_list %} {% if action_form and actions_on_top and
cl.full_result_count %}
{% admin_actions %} {% endif %} {% result_list_custom cl %} {% if action_form
and actions_on_bottom and cl.full_result_count %}
{% admin_actions %} {% endif %}
{% endblock %}
```

Ensuite, copiez le fichier change_list_result.html dans votre répertoire de templates, renommez-le en change_list_result_custom.html et modifiez-le comme bon vous semble.

En faire plus avec les vues : générer modèles et URL à la volée

Dans cet exemple d'application, nous allons voir comment générer des modèles et des URL à la volée, très simplement en profitant du dynamisme du langage Python. Pour cela, nous utiliserons une application de prototypage assez semblable à databrowse que vous avez découvert au début de cet ouvrage.

Une application de prototypage

Nous souhaitons profiter de la grande versatilité des vues génériques pour mettre en place un système de prototypage. L'idée est de pouvoir créer un modèle et de manipuler directement ses objets dans le navigateur sans avoir besoin de configuration complexe. À chaque fois que vous utilisez l'`admin` simplement pour ce type d'usage, c'est-à-dire vérifier si vos modèles correspondent bien à ce que vous en attendez, cette application de prototypage vous sera utile.

Voici son fonctionnement : pour chaque application présente dans les `INSTALLED_APPS`, on cherchera un fichier `scafold.py` qui devra contenir la variable globale `proto`. Celle-ci sera elle-même un tuple ou une liste contenant les modèles devant être exposés par l'application de protoypage. Cette dernière devra ensuite être en mesure de créer automatiquement et à la volée les URL et les vues correspondantes.

Les URL

La difficulté ici est de trouver un moyen d'écrire les URL à la volée en fonction d'un fichier `scafold.py` présent dans les `INSTALLED_APPS`. Dans un premier temps, il faut donc écrire un code qui va retrouver ce fichier. Instanciez la variable `urls` ; elle servira pour stocker les URL que vous aurez construites :

```
from django.conf import settings
urls = []
for app in settings.INSTALLED_APPS:
    prototype = "%s.scafold" % app
    try:
        a = __import__(prototype)
    except ImportError:
        a = None
```

Avec cette partie de code, pour chaque application, vous importez s'il existe le fichier `scafold.py`. Vous y recherchez la variable qui vous intéresse, à savoir `proto`, qui est une liste ou un tuple contenant les modèles utile pour créer les URL. Vous aurez besoin du nom de l'application et du nom du modèle en minuscules :

```
if a:
    for model in a.scafold.proto:
        root = model._meta.app_label,
        model._meta.object_name.lower()
```

La variable `root` peut maintenant être utilisée pour construire la première partie de l'URL, le *pattern* :

```
pattern = r"^%s/%s/$" % root
```

La part de magie va surtout résider dans des vues un peu spéciales que vous allez créer dans un instant. Pour le moment, partez du principe qu'elles existent déjà et créez vos URL :

Pour la liste des objets d'un modèle

```
liste = url(pattern,
    ProtoList.as_view(model=model),
    name="%s-list" % model._meta.object_name.lower())
    )
```

Pour le détail d'un objet

```
detail =url("%s/%s/(?P<pk>\d+)/" % root,
    ProtoDetail.as_view(model=model),
    name="%s-detail" % model._meta.object_name.lower()
    )
```

Ajoutez-les à votre liste d'URL :

```
urls.append(liste)
urls.append(detail)
```

Puis créez l'`urlpatterns` :

```
urlpatterns = patterns('prototype',)
for elem in urls:
    urlpatterns += patterns('',
                            elem)
```

Une fois cela fait, il faut encore créer les vues qui vont répondre à ces URL, ici `ProtoList` et `ProtoDetail`.

Les vues

Plusieurs problèmes se posent à vous. Vous devez unifier le nom des templates de façon à ce que deux templates, `object_list.html` et `object.html` soient utilisés quel que soit le modèle de départ. Vous devez générer les `absolute_urls` ainsi qu'une méthode renvoyant l'URL de la liste des objets. C'est assez complexe car cela demande d'avoir accès aux modèles. Vous allez donc créer des classes modèles à la volée ! Par conséquent, au lieu d'écrire :

```
class ProtoList(ListView):
    template_name = "object_list.html"
```

```
     def get_queryset(self) :
        return self.model
class ProtoDetail(DetailView) :
    template_name = "object.html"
    def get_queryset(self) :
        self.model
```

Vous devrez écrire :

```
class ProtoList(ListView):
    template_name = "object_list.html"

    def get_queryset(self):
        self.model = proxify_model(self.model)
        return self.model.objects.all()

class ProtoDetail(DetailView):
    template_name = "object.html"

    def get_queryset(self):
        self.model = proxify_model(self.model)
        return self.model.objects.all()
```

La différence ici est l'utilisation de la fonction `proxify_model` qui n'existe pas encore.

Les modèles

Cette fonction est une *factory*, c'est-à-dire que c'est une « usine à classes ». Elle va créer une classe à la volée et renvoyer une classe `model` qui va remplir vos besoins. Elle se situe dans `models.py` et voici à quoi elle ressemble :

```
def proxify_model(model):

    class ProxifiedModel(model, Prototypage):
        class Meta:
            proxy = True
    ProxifiedModel._meta = model._meta
    return ProxifiedModel
```

Ce que fait cette fonction est plus simple qu'il n'y paraît. Dans un premier temps, on définit une nouvelle classe `ProxifiedModel` qui hérite de la classe `model` que l'on a donnée en argument à la fonction et elle hérite également d'une autre classe `Prototypage` dont nous allons parler dans un instant.

Dans l'attribut `meta`, vous trouverez toutes les informations relatives à votre classe : nom de l'application, nom du modèle… Pour continuer à employer les données de la

classe originelle (celle que vous avez donnée en argument), les `meta` de `ProxifiedModel` sont une copie de ceux de la classe originelle.

Enfin, la classe `ProxifiedModel` est de type `proxy`. Les modèles de type `proxy` sont une copie d'une autre classe `model`. Il servent globalement à traiter un modèle d'une autre manière, avec un autre nom, d'autres méthodes, etc., sans qu'il n'y ait de création de table ou d'ajout de champ. La base de données continue à « voir » le modèle originel. Pour les besoin du moment, c'est parfait.

Reste donc cette fameuse classe `Prototypage` que nous n'avons pas encore écrite. Elle va devoir implémenter plusieurs méthodes :

- une renvoyant tous les champs et valeurs du modèles dans une liste, de façon à ce qu'il soit possible d'itérer dessus lors du rendu du template ;
- une renvoyant le nom du modèles ;
- une déterminant l'URL d'un objet unique ;
- une pour calculer l'URL de la liste d'objets.

```
class Prototypage(models.Model):

    def get_model(self):
        return self._meta.object_name

    @models.permalink
    def get_absolute_url(self):
        name = "%s-detail" % self._meta.object_name.lower()
        return (name, (), {"pk": self.pk})

    @models.permalink
    def get_list_url(self):
        return ("%s-list" % self._meta.object_name.lower(), (), {})

    def get_fields(self):
        return [
            (field.verbose_name, field.value_to_string(self))
            for field in self.__class__._meta.fields
            ]

    class Meta:
        abstract = True
```

La méthode `get_model` s'explique d'elle-même : vous allez chercher dans l'attribut `_meta` de la classe le nom de votre modèles. La méthode `get_absolute_url` va tout simplement recréer le `name` de l'URL correspondante, de façon à ce que le décorateur `@models.permalink` puisse la retrouver. La même façon de faire est employée pour la méthode `get_list_url` qui retourne la page listant tous les objets de vos modèles. La

méthode `get_fields` retourne une liste de tuples contenant chacun deux éléments : le nom du champ et sa valeur. Au moment d'écrire les templates, cela va grandement vous rendre service. Enfin, nous définissons cette classe de type `abstract`, c'est-à-dire qu'il n'y a pas de base de données créée pour cette classe. En réalité, elle n'a d'usage que pour Django. Elle est faite pour qu'une autre classe de type `model` en hérite.

Les templates

Maintenant que tout est prêt au niveau des modèles, des URL et des vues, il ne reste plus qu'à écrire les templates. Ici, il n'y a rien de complexe. Voici un exemple de templates, mais vous pouvez utiliser les vôtres, ajouter un peu de CSS et autres si vous le souhaitez :

```
<h1>{% with object_list|first as title %}{{title.get_model}}{%endwith%} List</
h1>
<ul>
{% for object in object_list %}
<li>{% include "include/object.html" %}
<a href="{{object.get_absolute_url}}">Details</a>
</li>

{% endfor %}
</ul>
```

Pour ne pas dupliquer le code qui correspond à l'affichage d'un objet unique et que l'on va retrouver sur les templates `object_list` et `object`, nous l'avons placé dans `include` ; nous y reviendrons. Pour le moment, sur le template `object_list`, la ligne qui mérite commentaire est la suivante :

```
<h1>{% with object_list|first as title %}{{title.get_model}}{%endwith%} List
```

Il s'agit simplement de prendre le premier élément de la liste et d'utiliser sa méthode `get_model` afin d'afficher le nom du modèle qui est consulté.

Le template `object` est sans doute plus simple encore et n'appelle pas de commentaires particuliers :

```
<h1>Detail of {{object.get_model}} {{object.pk}}</h1>
{% include "include/object.html" %}
<a href="{{object.get_list_url}}">List</a>
```

Il nous reste donc à vous montrer `include/object.html` qui est employé sur ces deux templates :

```
<ul>
{% for field, value in object.get_fields %}
```

```
<li>{{field}}: {{value}}</li>
{% endfor %}
</ul>
```

Il s'agit simplement d'itérer sur la méthode `get_fields` et d'afficher dans une liste les différents champs de chaque modèle.

Votre application est maintenant fonctionnelle. Il serait tout à fait possible, sur le même principe, d'implémenter les vues de création et de suppression d'objet pour obtenir une interface 100 % CRUD. Faites-le pour vous entraîner. Cela étant, nous vous déconseillons très fortement de mettre cette solution en place sur un site en production car vous aurez des problèmes de sécurité à gérer. De plus, si vous avez réellement besoin d'ajouter et de supprimer des objets, vous avez l'application `admin` de Django qui remplit à merveille ces besoins.

Aller plus loin avec les templates

JavaScript est un formidable outil pour générer des graphiques. Il est courant de récupérer des listes de données via Ajax, par exemple, et de les traiter en JavaScript. Seulement, vous êtes obligé de rendre la page en HTML et de faire une seconde requête vers le serveur pour obtenir les données dont JavaScript a besoin pour créer ses graphiques. Il est pourtant possible de se servir du moteur de templates de Django pour produire directement cette liste. Par exemple, si votre vue fournit une variable `graph_all` contenant les données, vous pouvez initialiser une variable JavaScript de la façon suivante :

```
var time_data_set = {
    all : {label:"all", data:{{graph_all}},
    points: { show: true },
    lines:{ show: true}, color:0 },
}
```

Vous pouvez même vous servir des boucles et conditions du système de templates pour remplir une liste JavaScript :

```
{% if hashtags %}
var all = new Array()
{% for hashtag in hashtags %}
{% ifchanged hashtag.name %}
all.push(["{{hashtag.name}}",{{hashtag.times}}])
{% endifchanged %}
{% endfor %}
{% endif %}
```

Utiliser Django en dehors de Django : déjouer l'anicroche

Au cours de votre découverte des différents éléments de Django, vous vous êtes sans doute dit que certaines parties de ce framework pourraient être utilisées en dehors de Django. On peut citer, par exemple, le moteur de templates ou les fonctions d'envoi d'e-mails.

Il est en effet tout à fait possible d'utiliser des parties de Django en dehors du framework, même si ce n'est pas aussi évident qu'il y paraît au départ. Nous allons prendre en exemple le moteur de templates, mais n'hésitez pas à essayer avec toutes les parties de Django auxquelles vous trouverez un intérêt en dehors d'un contexte web.

Commençons par ce qui pose problème, dans un premier temps avec un code très simple qui va lamentablement échouer :

```
$ python
>>> from django.template import Template, Context
>>> template = Template('{%for mot in phrase %} {{mot} {%endfor%}')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/yohann/Dev/WORK/TEST/apimc_env/lib/python2.7/site-packages/
django/template/base.py", line 123, in __init__
    if settings.TEMPLATE_DEBUG and origin is None:
  File "/home/user/projet/virt_env/lib/python2.7/site-packages/django/utils/
functional.py", line 184, in inner
    self._setup()
  File "/home/user/projet/virt_enlib/python2.7/site-packages/django/conf/
__init__.py", line 40, in _setup
    raise ImportError("Settings cannot be imported, because environment variable
%s is undefined." % ENVIRONMENT_VARIABLE)
ImportError: Settings cannot be imported, because environment variable
DJANGO_SETTINGS_MODULE is undefined.
```

Django ne trouve pas le fichier `settings.py` et ne peut donc pas employer des éléments de son framework dans un simple shell Python. Pourtant, il n'est pas concevable que vous deviez créer un fichier `settings.py` pour être en mesure d'utiliser le langage de templates. D'ailleurs, en regardant de plus prêt l'erreur qui vous est affichée, vous constaterez que Django ne se plaint pas de ne pas trouver les templates ; il se plaint que la variable `DJANGO_SETTINGS_MODULE` ne soit pas définie.

La fonction `django.conf.settings.configure()` vous permettra de la définir et de fournir à Django les éléments dont il a besoin pour être utilisé sans l'ensemble du framework. Cette fonction l'aide à trouver la fameuse variable qui vous manque. Vous pouvez donc vous servir librement le moteur de templates en dehors de Django.

Pour vous le démontrer, voici un petit script qui affiche les valeurs d'utilisation des disques durs sur un système Unix, Linux ou Mac OS X :

```
import os
from django.conf import settings
from django.template import Context, Template
settings.configure()
lines = os.popen("df -lh").read()
lines = lines.split()
step = 0
results = {'lines': []}
while len(lines) > step:
     results['lines'].append(lines[step:step+7])
     step +=7

template_string = "{%for line in lines %} {%for elem in line%} {{elem}}
|{%endfor%}\n {% endfor %}"
template = Template(template_string)
context = Context(results)
print template.render(context)
```

Ici, `settings.configure` ne contient aucune valeur, car aucun des settings n'est utile. En effet, nous avons employé que des classes et des méthodes de Django qui n'ont pas besoin de configuration particulière. Si, en revanche, nous avions souhaité enregistrer notre template dans un fichier, nous aurions dû préciser une valeur pour `TEMPLATE_DIR` afin que Django puisse trouver le template :

```
settings.configure(TEMPLATE_DIRS=('<chemin absolu vers les templates>',)
```

Index

__doc__ 83
__unicode__ 48, 194

A

admin
  change_list_result 387
  surclassement des templates 386, 387
  template 385
admin.py 45
administration
  __unicode__ 356
  activer l'interface 350
  afficher des champs 362
  delete_model 367
  documentation 354
  filtre 361
  formulaire 362, 367
  héritage 352
  HTML 359
  list_display 355
  maintenance 353
  ModelAdmin 366
  objets liés 364
  objets reliés 358
  ordonner les résultats 362
  personnaliser l'affichage 357
  pour les utilisateurs 354
  prototypage 350
  recherche 360
  save_model 366
  sécurité 353
  sélectionner les champs 357
agrégation 284
Ajax 156, 262
amélioration des URL 180
API REST 319, 326
  filtrer les ressources 330
application
  ajouter à settings.py 43
  ajouter à urls.py 44
application CRUD 164
arbre DOM 155
authentification 91, 103, 318, 321, 335
  access_token 321, 327
  activer un utilisateur 339
  authenticate 339
  backend 336
  connecter un utilisateur 339
  créer un utilisateur 338
  déconnecter un utilisateur 342
  groupe 347, 348
  login 339, 341
  login_required 344
  logout 342
  permission 345, 346, 348, 349
  permission_required 347
  RemoteUserBackend 336
  sécurité 338
  template 344
  User 337
  vérifier une connexion 343
autorisation 97, 321

B

backend
  console 196
  SMTP 195
base de données 25, 213, 236
  supprimer un enregistrement 129
bibliothèque
  csv 266
  json 263
  ReportLab 267
Bootstrap 150

C

carnet d'adresses 189
code HTTP 255
commentaire
  affichage 370
  ajout 369
  liste 369
  template 369
content-disposition 265
content-type 256, 262, 263, 265
contenttypes
  requête générique 373
Context 258
context 177
context_processors 149, 259
contexte 285
contrainte d'unicité 125, 237
contribs 335
  admin 350
  auth 335
  comments 368
  content-types 371
  message 375
  syndication 381
courriel avec Django 192, 195, 204
création d'un objet
  post_save 198
  signal 198
créer des tables 80
csrf_token 94
CSV (Comma Separated Values) 265

D

databrowse 53
datetime 170
décorateur 211, 273, 344, 347
décorateur Python 98
découplage de l'application 98
DEFAULT_SETTINGS 260
démarche Django 24
DetailView 173
développement dirigé par les tests 82
développement par itérations 20
dictionnaire GET 62
docstring 83
doctest 83
dumps 263

E

encodage 81

F

fichier statique 64
flatpage 374
flux RSS 381
formulaire 93, 100, 201
  {% csrf_token %} 302
  affichage 301, 312
  as_p 302
  as_table 302
  as_ul 302
  attrs 311
  création 116
  créer un validateur 305
  créer un widget 311
  enregistrement des données 102
  factorisation 178
  field 299, 302
  filtrer des choix 128
  filtrer les champs affichés 131
  flatatt 312
  Form 299
  form.cleaned_data 301
  help_text 307
  is_valid 301
  L'option required 303
  label 307
  ModelForm 115
  option d'affichage 307
  présélectionner un champ 127
  render 311
  supprimer un enregistrement 129
  traitement 300
  validateur de champ 303, 304, 306
  validation 101
  widget 299, 307, 308, 309
framework CSS 150
from local_settings import * 77

G

gestionnaire de versions 22
get_absolute_url 183
get_queryset 170

H

HttpRequest 253, 262
HttpResponse 35, 255, 257, 262

I

identifiant unique 154
include 146
initialisation d'un projet 13
  runserver 14
  startapp 23
  startproject 14
installation
  Cygwin 7
  Django 12
  éditeur de texte 5
  Mercurial 11
  pip 8
  Python 6
  setuptools 7
  virtualenv 8
INSTALLED_APPS 27
interface CRUD 46, 132, 275
interface d'administration 385
  modifier 47
internationalisation 26

J

Javascript 151
jQuery 151

K

kwargs 177

L

LESS 150
ListView 165, 194
login_required 273

M

manage.py 73
manager 242
  ajouter une méthode 243
  construire 243
  modifier une méthode 244
manager Python 123
mark_safe 293
message
  template 376
  vue 376
Meta 236
méthode agile 22
méthode de modèle 234
méthode de modèle prédéfinie 237
Mezzanine 317
middleware 268, 336, 337, 374
mixin 274
  DateMixin 276
  MultipleObjectMixin 275
mode de développement 24
modèle 39, 79, 260
ModelForm 115
MVC 29, 95

O

oauth2 318, 321
objet Q 247
ordre des résultats 236
ORM 217, 218
  créer un type de champ 223
  héritage Abstract 232
  héritage de table 232
  héritage multitable 234
  héritage Proxy 234
  option de champ 222
  relation entre tables 225
  relation foreign key 228
  relation ManyToMany 230
  relation OneToOne 226, 227
  type de champ 221

P

paginate_by 277
pagination 167
paginator 168
permission 237

personnalisation du contenu 95
pip 72
principe de programmation
  DRY (Don't Repeat Yourself) 13
prototypage 389
  modèle 391
  template 393
  URL 389
  vue 390
provider 318
proxify_model 391
python manage.py shell 41

Q

queryset 55, 128, 170, 218, 239, 260, 263, 272, 275
  coût 241
  description 239
  relation 240

R

reconstruction d'URL 181
relation many-to-many 80, 119
render 67, 259
render_to_response 58, 65, 257
request 34, 62, 252, 258, 259, 268
RequestContext 66, 258
requests 325
requête
  Ajax 262
  en-tête 254
  GET 253
  méthode 253
  POST 253
  workflow 252
requêtes fréquentes 191
requirements.txt 72
response 255, 268
reverse 181, 194
revue de code 176

S

scafold.py 389
send_mail 196
sérialisation 263, 264, 265
serializer 263
settings.py 24, 76, 260, 268, 336, 350, 369, 374, 375, 377, 379, 381, 386, 387, 395
SGBD
  NoSQL 215
  relationnel 213
sitemap 377
  raccourci 378
sites
  lier un contenu 380
  site courant 380
Slumber 328
slumber 324
south 76, 81
startapp 73
startproject 73
STATIC_URL 65
staticfiles 65
syncdb 28
syndication 381

T

tastypie 319
TDD (Test Driven Developpement) 82
template 56, 141, 294
  boucle 287
  branchement conditionnel 287
  compilation 256
  context_processors 149
  contexte 285
  créer un filtre 291, 292
  créer un templatetag 291, 294
  de base 142
  décorateur 296
  django.template.Node 294
  extends 144
  fichier statique 149
  filtre 289
  gestion 166
  images 149
  include 146
  includes 291
  langage 285
  organisation 289

pagination 167
parser 294
pliage/dépliage 153
register 292
styles CSS 149
tag avec arguments 296
template.Library 292
templatetag 289
token 294
token.split_content() 294
template_name 182, 275
TemplateView 96, 166
test unitaire 87, 88, 103
d'échec 108
des formulaires 106
des URL 104
through 80
toggle 155
Twitter 150
type MIME
application/json 262
application/pdf 267
text/csv 265
text/html 255, 256

U

unique 125
unique_together 126
URI (Uniform Resource Identifier) 315
URL (Uniform Resource Locator) 315
urls.py 30, 60
User 337

V

verbe REST 316
DELETE 315, 317, 330
GET 315
HEAD 317
PATCH 315, 317
POST 315
PUT 315, 317, 328
views.py 32, 55, 261
virtualenv 71
vue 251, 261
héritage 279
middleware 268
pagination 168
vue générique 96, 163, 175, 176, 182, 269, 389
ArchiveIndexView 277
BaseDateListView 277
BaseListView 277
CreateView 182, 275, 276
DayArchiveView 278
décorateur 273
DeleteView 183, 275
DetailView 173, 176, 271
ListView 271, 272
MonthArchiveView 278
TemplateView 166, 269
TodayArchiveView 278
UpdateView 183, 205
verbe REST 279
vue-classe 278, 280
WeekArchiveView 278
YearArchiveView 277